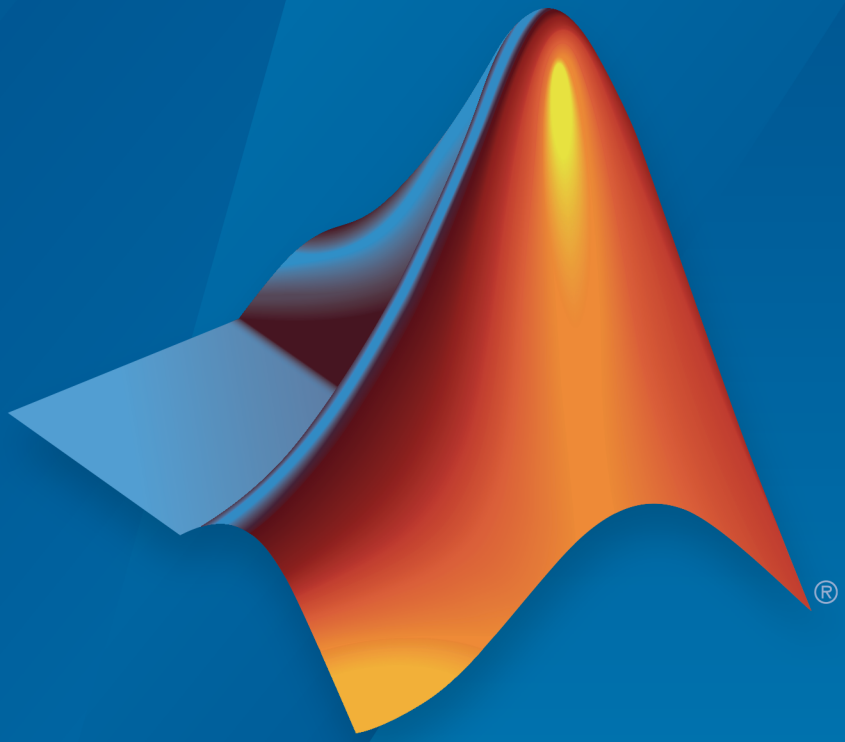
Embedded Coder®

User's Guide



MATLAB® & SIMULINK®

R2017a

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

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

Embedded Coder® User's Guide

Revision History

April 2011	Online only	New for Version 6.0 (Release 2011a)
September 2011	Online only	Revised for Version 6.1 (Release 2011b)
March 2012	Online only	Revised for Version 6.2 (Release 2012a)
September 2012	Online only	Revised for Version 6.3 (Release 2012b)
March 2013	Online only	Revised for Version 6.4 (Release 2013a)
September 2013	Online only	Revised for Version 6.5 (Release 2013b)
March 2014	Online only	Revised for Version 6.6 (Release 2014a)
October 2014	Online only	Revised for Version 6.7 (Release 2014b)
March 2015	Online only	Revised for Version 6.8 (Release 2015a)
September 2015	Online only	Revised for Version 6.9 (Release 2015b)
October 2015	Online only	Rereleased for Version 6.8.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 6.10 (Release 2016a)
September 2016	Online only	Revised for Version 6.11 (Release 2016b)
March 2017	Online only	Revised for Version 6.12 (Release 2017a)

Check Bug Reports for Issues and Fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

Contents

Model Architecture and Design

1 Modeling Environment for Embedded Coder

2 Modeling in Simulink Coder

3 Subsystems in Simulink Coder

4 Code Generation of Functions and Function Callers in Simulink Coder

5 Referenced Models in Simulink Coder

6 Combined Models in Simulink Coder

7 Configure Model Parameters for Simulink Coder

8 Model Protection in Simulink Coder

9 Component Initialization, Reset, and Termination in Simulink Coder

10 Stateflow Blocks in Simulink Coder

11 Block Authoring and Code Generation for Simulink Coder

12 Guidelines and Standards for Embedded Coder

13 Patterns for C Code in Embedded Coder

14 Variant Systems in Embedded Coder

15 Timers in Simulink Coder

16 Time-Based Scheduling in Simulink Coder

17 Event-Based Scheduling in Simulink Coder

18 Scheduling Considerations in Embedded Coder

Data, Function, and File Definition

19 Data Representation in Simulink Coder

20 Data Definition and Declaration Management in Embedded Coder

21 Data Types in Embedded Coder

22 Module Packaging Tool (MPT) Data Objects in Embedded Coder

23 Custom Storage Classes in Embedded Coder

24 Data Object Wizard in Embedded Coder

25 Entry-Point Functions and Scheduling in Simulink Coder

26 Function and Class Interfaces in Embedded Coder

27 Memory Sections in Embedded Coder

Code Generation

28 Configuration for Simulink Coder

29 Configuration in Embedded Coder

30 System Target File Configuration

33 Source Code Generation in Simulink Coder

34 Source Code Generation in Embedded Coder

35 Report Generation in Embedded Coder

36 Code Appearance in Embedded Coder

37 Code Replacement in Simulink Coder

38 Code Replacement for Simulink Models in Embedded Coder

39 External Code Integration in Simulink Coder

Program Building, Interaction, and Debugging in Simulink Coder

40

41 Host/Target Communication in Simulink Coder

42 Logging in Simulink Coder

43 Data Interchange Using the C API in Simulink Coder

44 ASAP2 Data Measurement and Calibration in Simulink Coder

45 Direct Memory Access to Generated Code for Simulink Coder

46 Desktops in Simulink Coder

47 Desktops in Embedded Coder

48 Real-Time Systems in Simulink Coder

49 Real-Time and Embedded Systems in Embedded Coder

50 Export Code Generated from Model to External Application in Embedded Coder

51 Code Replacement Customization for Simulink Models in Embedded Coder

52 Code Replacement Customization for MATLAB Code

Performance

Optimizations for Generated Code in Simulink Coder

53

54 Configuration in Embedded Coder

55 Data Copy Reduction in Embedded Coder

56 Execution Speed in Embedded Coder

57 Memory Usage in Embedded Coder

58 Code Execution Profiling in Embedded Coder

59 Code Execution Profiling for MATLAB Coder

Verification

60 Simulation and Code Comparison in Simulink Coder

61 Code Tracing in Embedded Coder

62 Component Verification in Embedded Coder

63 Component Verification With a Real-Time Target Environment in Embedded Coder

64 Numerical Equivalence Checking in Embedded Coder

65 Numerical Consistency between Model and Generated Code

66 Software-in-the-Loop Execution for MATLAB Coder

67 Code Coverage in Embedded Coder

Embedded IDEs and Embedded Targets

68 Getting Started with Embedded Targets in Embedded Coder

69 Run-Time Data Interface Extensions in Simulink Coder

70 Build Process Integration in Simulink Coder

71 Custom Target Development in Simulink Coder

72 Project and Build Configurations for Embedded Targets in Embedded Coder

73 Verification and Profiling Generated Code in Embedded Coder

74 Processor-Specific Optimizations for Embedded Targets in Embedded Coder

Build Configuration for Code Generation from MATLAB Code

75

76 Code Replacement for MATLAB Code

77 Storage Classes for Code Generation from MATLAB Code

Verification of Code Generated from MATLAB Code

78

Model Architecture and Design

1

Modeling Environment for Embedded Coder

Design Models for Generated Embedded Code Deployment

When using Embedded Coder® to generate code for an embedded system architecture, it is important to design your Simulink models with code generation in mind from the very beginning of the design process. Think about relevant design factors and issues such as:

In this section...

Application Algorithms and Run-Time Environments

Use Simulink to design models that represent application algorithms and run-time environments from which you intend to generate deployable code. Depending on your application, you might deploy code to an execution environment that consists of a combination of:

Execution Environment Components	Choices
Hardware	• Development computer • Rapid-prototyping board • Microprocessor • Microcontroller • FPGA • ASIC
Cores	• Single • Multiple
Operating system	• General-purpose • Real-time • None (bare metal)
Scheduling	• Single-tasking

Execution Environment Components	Choices
	• Multitasking • Interrupt driven • Concurrency • Provided by operating system • Generated from model
Application algorithm code	• Generated from model • External code

As you design models to generate C or C++ code for rapid prototyping or production deployment, keep in mind the execution environment. Generate code that meets implementation requirements and avoids potential design rework. As the preceding table reflects, the execution environment for code that you generate can range from relatively simple to complex. For example, a simple case is code that you generate from a single, single-tasking model that runs on a single-core microprocessor. A complex case is code that generate from a model partitioned to run as a distributed system on a multicore microprocessor and an FPGA.

Software Execution Framework for Generated Code

Part of an application execution environment is the software execution framework that is responsible for scheduling and running the generated code. That software can preexist, as in the case of an operating system and its scheduler, or you can code the software manually. The level of complexity varies depending on which of the following modeling and code generation scenarios applies:

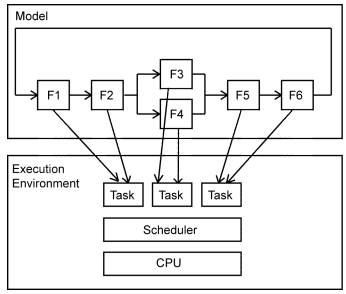
- Generate code from a single top model, which represents the algorithms intended to run in the execution environment.
- Generate code from a model, which represents part of an overall algorithm. You can mix the generated code with code written manually and code generated from other sources or releases of MathWorks® products.

Single Top Model

For a single top model, the software execution framework is responsible for running generated code the same way that Simulink simulates the model. Functions in the generated code are highly coordinated and optimized because Simulink is aware of

dependencies. The framework interfaces with code generated for the top model only. Code generated for a top model handles interfacing with code for referenced Model blocks.

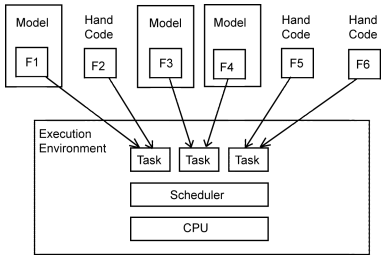
Consider the following example, where a single top model is mapped to tasks that run on a single-core CPU.



For this system, you map model clock rates to tasks that run on the hardware. You can choose for Simulink to map the rates implicitly or you can map them explicitly in your model. You can model latency effects resulting from how you map rates in a model to single-tasking or multitasking execution environments. Simulink schedules the tasks properly based on rates in the model and data dependencies between tasks. The code generator implements the same dependencies in the code that it generates. The software execution framework invokes generated entry-point functions at rates based on system timers and interrupts. The generated code executes in the same manner that Simulink simulates the model, and contains code dedicated to communicating data between functions running at different rates.

Multiple Top-Level Models

When you generate code from multiple top models separately and mix that code with code acquired in other ways, the execution environment of the application takes on more software execution framework responsibility. For this modeling scenario, you generate code for standalone, atomic reusable components.



With this scenario, Simulink is not aware of model dependencies. Functions in code generated from the different models are minimally coordinated and optimized. For example, the models might share generated utility functions. Potential optimizations that cross model boundaries are not possible. You must design the software execution framework taking into account dependencies between units of code, including execution order. For an application that requires concurrent execution across multiple cores, you must consider data latency effects across the cores.

The code generator helps you address software execution framework challenges, such as sharing global data and avoiding identifier conflicts. The code generated for a each model handles the interfacing for referenced Model blocks.

Map Embedded System Architecture to Simulink Modeling Environment

When mapping an embedded system architecture to the Simulink modeling environment, think about the model design.

"Modeling Algorithms" on page 1-6	Given initial state and input, a set of tasks or instructions that efficiently produce the results that you want.
"Modeling Interfaces" on page 1-6	Mechanisms that enable algorithm components to communicate and exchange information across component boundaries.
"Modeling Systems" on page 1-8	Collection of algorithm components that achieve a higher-level, domain-specific goal or result. Components often share resources.
"Modeling Run-Time Environments" on page 1-11	Framework that handles scheduling of system algorithm resources and execution.

Consider the following questions regarding an embedded system architecture with corresponding modeling capabilities and links to related information. Use the information as a guide for mapping your architecture details to the Simulink modeling environment. Designing a model architecture with your specific embedded system architecture in mind can help you avoid rework and future conversion and maintenance costs.

Modeling Algorithms

Architecture Considerations	Modeling Considerations	Related Information
What is the system domain?	Product prerequisites (based on domains of components)	• "Supported Products and Block Usage" (Simulink Coder) • "Simulink Control Design" • "Model Signal Processing Systems" (Simulink) • "Signal Generation, Manipulation, and Analysis" (DSP System Toolbox)
Does the system involve physical domains, such as mechanical, electrical, or hydraulic domains?	Physical systems	• "Model Physical Systems" (Simulink) • "Basic Principles of Modeling Physical Networks" (Simscape) • "Essential Physical Modeling Techniques" (Simscape)
What aspects of your algorithm can you represent with blocks provided by MathWorks products? What blocks do you need to create?	Block usage, creation, and customization	• "Supported Products and Block Usage" (Simulink Coder) • "External Code Integration" (Simulink Coder)
Does the architecture include state machine components?	Event-driven system	"Basic Approach for Modeling Event-Driven Systems" (Stateflow)

Modeling Interfaces

Architecture Considerations	Modeling Considerations	Related Information
• What data must you represent in the generated code? • How do you need to represent input and output—data type, dimension, complexity? • Do the algorithms use floating-point or fixed-point arithmetic? • How will the data change?	Data representation	• "Interface Design" (Simulink) • "Data Representation" • "Modeling Patterns for C Code" • "Fixed-Point Designer"

Architecture Considerations	Modeling Considerations	Related Information
Where and how is data pulled into the system and pulled within the system?	Input	• "Comparison of Signal Loading Techniques" (Simulink) • "Modeling Patterns for C Code"
• Where and how is data pushed within the system and out of the system? • What external triggers are required?	Output	• "Simulation Data Inspector in Your Workflow" (Simulink) • "Control Data Representation by Applying Custom Storage Classes" on page 23-58
• What functions do you need to define for each component? • What is the prototype for each entry-point function?	Functions and function calls	• "Function and Class Interfaces" • "Function Prototyping" on page 13-67
Do you need to export functions that are invoked by controlling logic that is outside the model?	Function export	• "Export-Function Models" (Simulink) • "Generate Component Source Code for Export to External Code Base" on page 39-51
Does the system monitor signals or log data (for example, for calibration)?	C API and ASAP2 data exchange interfaces	• "Exchange Data Between Generated and External Code Using C API" (Simulink Coder) • "Export ASAP2 File for Data Measurement and Calibration" (Simulink Coder)
Do you need to replace code generated for functions or operators, for example, to optimize the code for specific hardware?	Code replacement	• "What Is Code Replacement?" (Simulink Coder) • "What Is Code Replacement Customization?" on page 51-3

Architecture Considerations	Modeling Considerations	Related Information
Do you need to control the placement of data or functions in memory?	Memory sections	• "Introduction to Custom Storage Classes" on page 23-2 • "Control Data and Function Placement in Memory by Inserting Pragmas" on page 27-2 • "Declare Constant Data as Volatile Using Memory Sections" on page 27-19
Is there a requirement for elaboration and future considerations?	Elaboration and future considerations	• "Interface Design" (Simulink)

Modeling Systems

Architecture Considerations	Modeling Considerations	Related Information
• What is the scope of the system? Controller? External environment or plant? Test harness? • How is the system partitioned into algorithm components (chunks of logic)? • Which components can you represent in Simulink? • Can you design components for reuse? What is the motivation for reuse (for example, division of labor or plug-n-play)?	Componentization	• "Interface Design" (Simulink) • "Componentization Guidelines" (Simulink) • "Design Partitioning" (Simulink) • "Custom Libraries and Linked Blocks" (Simulink) • "Export-Function Models" (Simulink) • "Custom MATLAB Algorithms" (Simulink) • "Code Generation of Subsystems" (Simulink Coder) • "Code Generation of Referenced Models" (Simulink Coder) • "Code Generation of Stateflow Blocks" (Simulink Coder)
• Do aspects of the system require unit testing?	Model referencing	• "Overview of Model Referencing" (Simulink)

Architecture Considerations	Modeling Considerations	Related Information
• Is a team of people collaborating on the project? • Do you need to protect intellectual property?		• "Componentization Guidelines" (Simulink) • "Code Generation of Referenced Models" (Simulink Coder) • "Generate Reusable Code for Unit Testing" (Simulink Coder)
Are you modeling a client-server architecture?	Simulink Function and Caller blocks	• "Diagnostics Using a Client-Server Architecture" (Simulink) • "Simulink Functions" (Simulink)
Is relevant legacy or custom code available?	External code integration	"External Code Integration" (Simulink Coder)
Can you apply a reference architecture or reference components?	Model and project templates	• "Create a Template from a Model" (Simulink) • "Create a New Project Using Templates" (Simulink)
Do you need to export functions that are invoked by controlling logic that is outside a model?	Export-function models	"Export-Function Models" (Simulink)
Is there a need to package the source code for a component as a shared object library to simplify distribution or sharing?	Shared object libraries (dynamic link libraries)	"Package Generated Code as Shared Libraries" on page 47-2

Architecture Considerations	Modeling Considerations	Related Information
Can you reuse functions?	Function reuse	• "Code Reuse For Subsystems Shared Across Models" (Simulink Coder) • "Reusable Library Subsystem" (Simulink Coder) • "Generate Reentrant Code from Top-Level Models" (Simulink Coder) • "Reusable Code and Referenced Models" (Simulink Coder) • "Generate Reusable Code for Atomic Subcharts" (Simulink Coder)
• Do components need to share access to global data? • Within the system, do state changes occur? In each case, how does the result get communicated? • Are there identifier (naming) issues to consider?	Shared data	• "Local and Global Data Stores" (Simulink) • "Default Data Structures in the Generated Code" (Simulink Coder) • "Storage Classes for Signals Used with Model Blocks" (Simulink Coder) • "Shared Constant Parameters for Code Reuse" (Simulink Coder) • "Data Stores in Generated Code" (Simulink Coder) • "Create Data Objects for Code Generation with Data Object Wizard" on page 24-2 • "Place Global Data Declarations and Definitions in Separate Files" on page 20-3 • "Customize Generated Identifier Naming Rules" on page 36-15

Architecture Considerations	Modeling Considerations	Related Information
Do you need to control placement of data or functions in memory?	Memory sections	• "Introduction to Custom Storage Classes" on page 23-2 • "Control Data and Function Placement in Memory by Inserting Pragmas" on page 27-2 • "Declare Constant Data as Volatile Using Memory Sections" on page 27-19
Are you required to apply the AUTOSAR standard? If yes, what aspects of the architecture involve AUTOSAR?	AUTOSAR	"AUTOSAR"
Does your system need to meet other standards or guidelines?	Standards and guidelines	"Support for Standards and Guidelines" on page 12-2

Modeling Run-Time Environments

Architecture Considerations	Modeling Considerations	Related Information
• What level of control over run-time interfacing does your application require? • How much of your system can you represent in a model?	Runtime interfacing	• "Execution of Code Generated from a Model" (Simulink Coder) • See Modeling Interfaces.
Is the system partitioned into concurrent components to maximize parallelism? Which components?	Concurrency	"Multicore Processor Targets" (Simulink)
• Are components driven by an external clock? • What clock rates do system components use? • Do components use a single rate or multiple rates?	Clocks and clock rates	"Interface Design" (Simulink)

Architecture Considerations	Modeling Considerations	Related Information
• Are components in the system driven by clocks? • What clock rates do system components use? • Do components use a single rate or multiple rates? • What are the priorities of system tasks and functions?	Time-based scheduling	• "Absolute and Elapsed Time Computation" (Simulink Coder) • "Time-Based Scheduling" (Simulink Coder)
• Are components in the system driven by events (interrupts)? • What are the priorities of system tasks and functions?	Event-based scheduling	• "Absolute and Elapsed Time Computation" (Simulink Coder) • "Event-Based Scheduling" (Simulink Coder) • "Basic Approach for Modeling Event-Driven Systems" (Stateflow)
Does the system need to handle initialization, reset, or terminate events?	Initialization, reset, termination	• "Create Model to Initialize, Reset, and Terminate State" (Simulink) • "Generate Code That Responds to Initialize, Reset, and Terminate Events" (Simulink Coder)
• Is the system a single-tasking or multitasking system? • Are components required to execute in real time? • What are the execution order dependencies (sequencing) between components? • What are the time constraints for task and function execution?	Task execution	• "Execution of Code Generated from a Model" (Simulink Coder) • "Modeling for Single-Tasking Execution" (Simulink Coder) • "Modeling for Multitasking Execution" (Simulink Coder)

Architecture Considerations	Modeling Considerations	Related Information
• If you know the processing platform, what is it? • Will the system run on a single-core or multicore processor? • Is the system a distributed system? • Is the processing platform hybrid or heterogeneous? • Does the architecture employ symmetric or asymmetric multiprocessing? If asymmetric, how is the platform software partitioned across CPUs?	Processing platforms	"Multicore Processor Targets" (Simulink)
• Do you want to generate and run a standalone executable that does not require an external real-time kernel or operating system? • Is a real-time operation system (RTOS) required? If yes, what RTOS?	Kernel, operating system	• "Deploy Generated Standalone Executable Programs To Target Hardware" on page 49-2 • "Deploy Generated Component Software to Application Target Platforms" on page 49-22

Model Templates for Code Generation

The code generator provides a set of built-in templates to use as a starting point to create models for common application designs. Use the templates to create models that are preconfigured to generate code for rapid-prototyping or embedded system applications.

Template	Description
Code Generation System	Basic model consisting of an Inport block and Output block.
Exported functions	Model for generating code from function-call subsystems. You can export each function-call subsystem separately by right-clicking a

Template	Description
	subsystem, selecting **C/C++ Code** > **Export Functions**, and clicking **Build**.
Fixed-step, multirate	Fixed-step model that uses multiple rates and consists of Inport blocks, an Outport block, and a Sum block. The model is configured to use a fixed-step discrete solver and to use two rates with **Periodic sample time constraint** set to `Unconstrained` and the **Treat each discrete rate as a separate task** option selected. Simulink inserts a Rate Transition block to handle the two sample rates.
Fixed-step, single rate	Fixed-step model that uses a single rate and consists of Inport blocks, an Outport block, and a Sum block. The model is configured to use a fixed-step discrete solver.

To create a model from a template:

1. On the MATLAB® home tab, click **Simulink**.
2. In the Simulink start page, expand **Embedded Coder**.
3. Select a template.
4. Click **Create**. A new model that uses the template contents and settings appears in the Simulink Editor window.

For more information, for example to create and use a template as a reference design, see "Create a Template from a Model" (Simulink).

More About

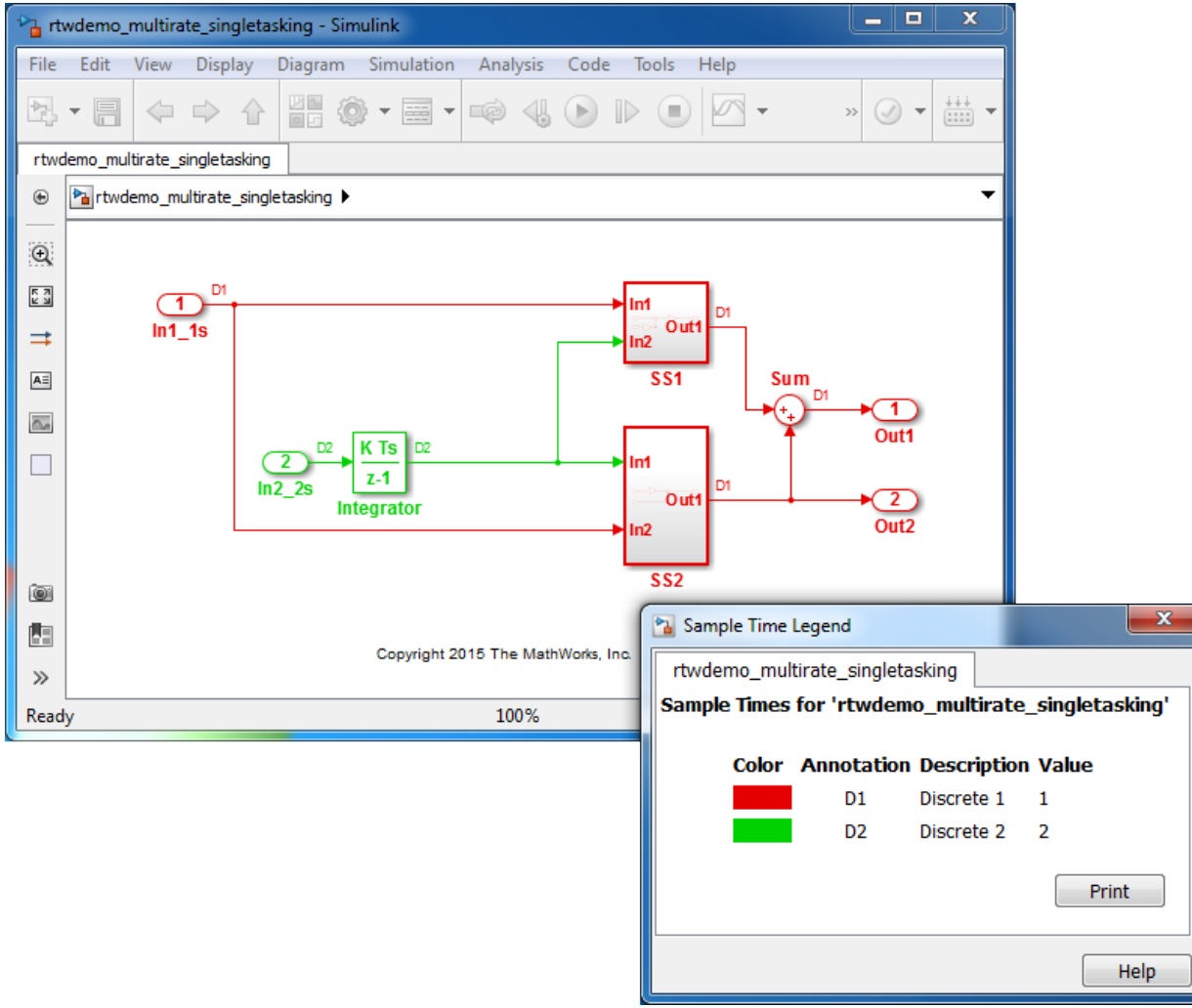
- "Model Single-Core, Single-Tasking Platform Execution" on page 1-15
- "Model Single-Core, Multitasking Platform Execution" on page 1-20
- "Model Concurrent Execution for Symmetric Multicore CPU Platforms" on page 1-25
- "Model Explicit Function Invocation with Atomic Subsystems" on page 1-33
- "Model Explicit Function Invocation with Function-Call Subsystems" on page 1-38
- "Model for AUTOSAR Platform" on page 1-42

Model Single-Core, Single-Tasking Platform Execution

This example shows a model designed and configured for embedded system code generation intended to execute on a single-core, single-tasking platform. The application algorithm is captured in a single model hierarchy, making it possible to use Simulink® time-based, single-task scheduling to simulate the model and execute the generated code.

Periodic Multirate Model Set Up for Single-Tasking Execution

Open the example model `rtwdemo_multirate_singletasking`. The model is configured to display color-coded sample times with annotations. To see them, after opening the model, update the diagram by pressing **Ctrl+D**. To display the legend, press **Ctrl+J**.



- Sample times for Inport blocks `In1_1s` and `In2_2s` are set to 1 and 2 seconds, respectively.
- To provide clean partitioning of rates, sample times for subsystems `SS1` and `SS2` are set to 1.

Relevant Model Configuration Parameter Settings

- **Solver > Type** set to `Fixed-step`.
- **Solver > Solver** set to `discrete (no continuous states)`.
- **Solver > Treat each discrete rate as a separate task** cleared.

Scheduling

Simulink® simulates the model based on the model configuration. Code generated from the model implements the same execution semantics. Simulink propagates and uses the Inport block sample times to order block execution based on a single-core, single-tasking execution platform.

For this model, the sample time legend shows an implicit rate grouping. Red represents the fastest discrete rate. Green represents the second fastest discrete rate.

The generated code schedules subrates in the model. In this example, the rate for Inport block `In2_2s`, the green rate, is a subrate. The generated code properly transfers data between tasks running at the different rates.

Benefits of implicit rate grouping:

- Simulink does not impose architectural constraints on the model.
- Your execution framework does not require details about underlying function scheduling and data transfers between rates. Therefore, the model interface requirements are simplified. The execution framework uses generated interface code to write input, call the model step function, and read output.
- The code generator optimizes code across rates based on single-tasking execution semantics.

Your execution framework can communicate with external devices for reading and writing model input. For example, model external devices by using Simulink S-Function blocks. Generate code for those blocks with the rest of the algorithm.

Generate Code and Report

Generate code and a code generation report. The example model generates a report.

Review Generated Code

From the code generation report, review the generated code.

- `ert_main.c` is an example main program (execution framework) for the model. This code controls model code execution by calling the entry-point function `rtwdemo_multirate_singletasking_step`. Use this file as a starting point for coding your execution framework.
- `rtwdemo_multirate_singletasking.c` contains entry points for the code that implements the model algorithm. This file includes the rate scheduling code.
- `rtwdemo_multirate_singletasking.h` declares model data structures and a public interface to the model entry points and data structures.
- `rtwtypes.h` defines data types, structures, and macros that the generated code requires.

Code Interface

Open and review the Code Interface Report. Use the information in that report to write the interface code for your execution framework:

1. Include the generated header file by adding directive `#include rtwdemo_multirate_singletasking.h`.
2. Write input data to the generated code for model Inport blocks.
3. Call the generated entry-point functions.
4. Read data from the generated code for model Outport blocks.

Input ports:

- `rtU.In1_1s` of data type `real_T` with dimension of 1
- `rtU.In2_2s` of data type `real_T` with dimension of 1

Entry-point functions:

- Initialization entry-point function, `void rtwdemo_multirate_singletasking_initialize(void)`. At startup, call this function once.
- Output and update entry-point (step) function, `void rtwdemo_multirate_singletasking_step(void)`. Call this function periodically at the fastest rate in the model. For this model, call the function every second. To achieve real-time execution, attach this function to a timer.

Output ports:

- `rtY.Out1` of data type `real_T` with dimension of 1

- `rtY.Out2` of data type `real_T` with dimension of 1

More About

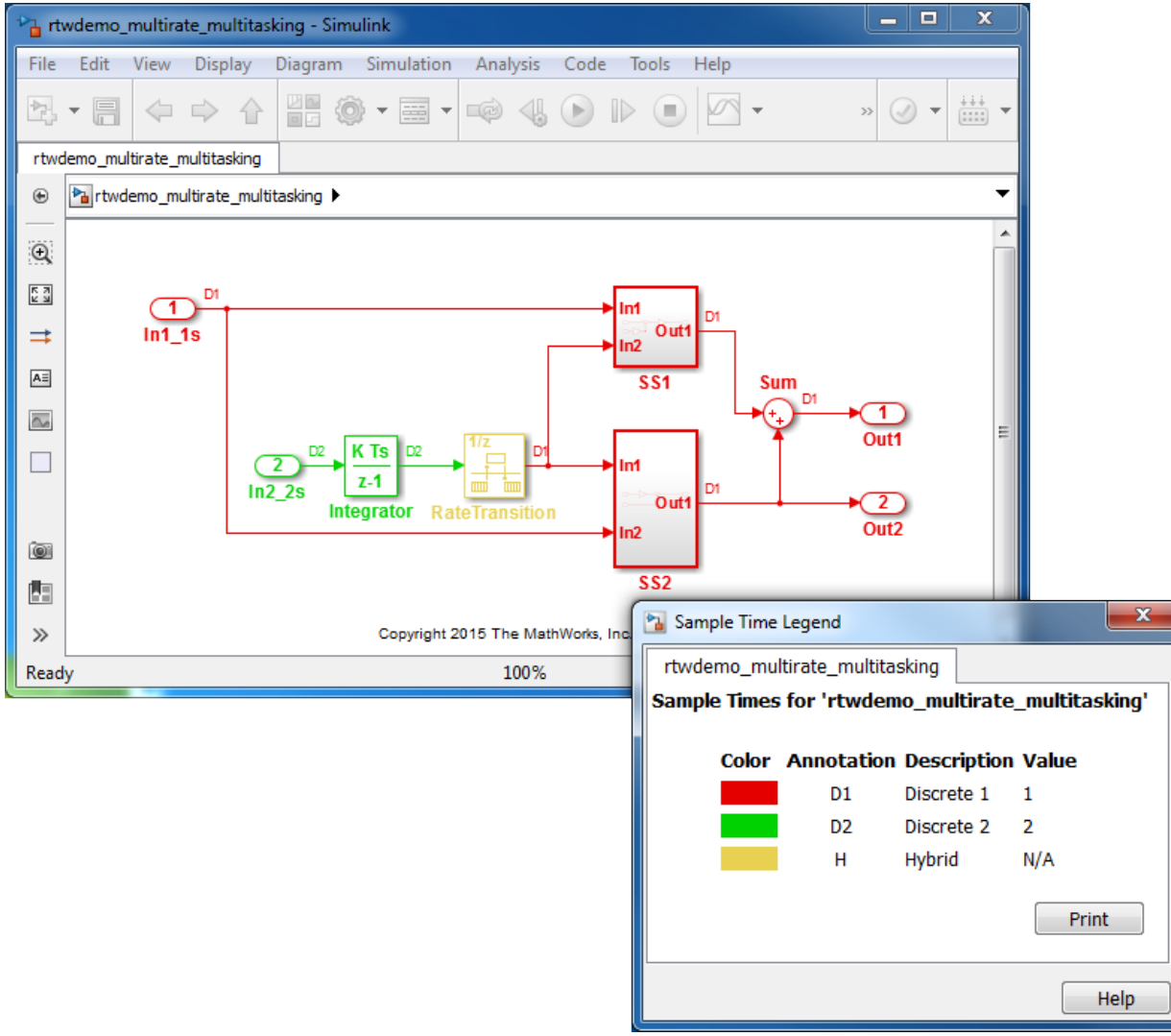
- "Modeling for Single-Tasking Execution" (Simulink Coder)
- "Deploy Generated Standalone Executable Programs To Target Hardware"
- "Customize Code Organization and Format"

Model Single-Core, Multitasking Platform Execution

Use Simulink® time-based, multitask scheduling to simulate and generate code for an application algorithm captured in a single model hierarchy. The model is designed and configured for an embedded system intended to execute on a single-core, multitasking platform. The model simulates and the generated code executes based on the model configuration and a rate monotonic scheduling algorithm.

Periodic Multirate Model Set Up for Multitasking Execution

Open the example model `rtwdemo_multirate_multitasking`. The model is configured to display color-coded sample times with annotations. To see them, after opening the model, update the diagram by pressing **Ctrl+D**. To display the legend, press **Ctrl+J**.



- Sample times for Inport blocks `In1_1s` and `In2_2s` are set to 1 and 2 seconds, respectively.
- To provide a clear partitioning of rates, sample times for subsystems `SS1` and `SS2` are set to 1.

- The Rate Transition block models an explicit rate transition. Alternatively, instruct Simulink to insert Rate Transition blocks for you by selecting model configuration parameter **Solver > Automatically handle rate transition for data transfer**.

Relevant Model Configuration Parameter Settings

- **Solver > Type** set to `Fixed-step`.
- **Solver > Solver** set to `discrete (no continuous states)`.
- **Solver > Treat each discrete rate as a separate task** selected.

Scheduling

Simulink® simulates the model based on the model configuration. Code that this model generates implements the same execution semantics. Simulink propagates and uses the Inport block sample times to order block execution based on a single-core, multitasking execution platform.

For this model, the sample time legend shows an implicit rate grouping. Red represents the fastest discrete rate. Green represents the second fastest discrete rate. Yellow represents the mixture of the two rates.

The generated code schedules subrates in the model. In this example, the rate for Inport block `In2_2s`, the green rate, is a subrate. The generated code properly transfers data between tasks that run at the different rates.

Benefits of implicit rate grouping:

- Simulink does not impose architectural constraints on the model. Create a model without imposing software architecture constraints within the model.
- Your execution framework does not require details about underlying function scheduling and data transfers between rates. Therefore, the model interface requirements are simplified. The execution framework uses generated interface code to write input, call the model step function, and read output.
- The code generator optimizes code across rates based on multitasking execution semantics.

Simulink enforces data transfer constraints to achieve rate monotonic scheduling:

- Data transfers occur between a single read task and a single write task.
- When data transfers between two tasks, only one task can preempt the other task.

- For periodic tasks, a task with a faster rate has a higher priority than a task with a slower rate. In addition, a task with the faster rate, preempts a task with a slower rate.
- Tasks run on a single processor.
- Time slicing, use of a defined time period during which a task can run in a preemptive multitasking system, is not allowed.
- Processes do not crash or restart, especially during data transfers between tasks.
- Read and write operations on byte-sized variables are atomic.

Your execution framework communicates with external devices for reading and writing model input. For example, model external devices by using Simulink S-Function blocks. Generate code for those blocks with the rest of the algorithm.

Generate Code and Report

Generate code and a code generation report. The example model generates a report.

Review Generated Code

From the code generation report, review the generated code.

- `ert_main.c` is an example main program (execution framework) for the model. This code controls model code execution by calling the entry-point functions `rtwdemo_multirate_multitasking_step0` and `rtwdemo_multirate_multitasking_step1`. Use this file as a starting point for coding your execution framework.
- `rtwdemo_multirate_multitasking.c` contains entry points for the code that implements the model algorithm. This file includes the rate scheduling code.
- `rtwdemo_multirate_multitasking.h` declares model data structures and a public interface to the model entry points and data structures.
- `rtwtypes.h` defines data types, structures, and macros that the generated code requires.

Code Interface

Open and review the Code Interface Report. Use the information in that report to write the interface code for your execution framework:

1. Include the generated header file by adding directive `#include rtwdemo_multirate_singletasking.h`.

2 Write input data to the generated code for model Inport blocks.

3 Call the generated entry-point functions.

4 Read data from the generated code for model Outport blocks.

Input ports:

- `rtU.In1_1s` of data type `real_T` with dimension of 1
- `rtU.In2_2s` of data type `real_T` with dimension of 1

Entry-point functions:

- Initialization entry-point function, `void rtwdemo_multirate_multitasking_initialize(void)`. At startup, call this function once.
- Output and update entry-point (step) function, `void rtwdemo_multirate_multitasking_step0(void)`. Call this function periodically at the fastest rate in the model. For this model, call the function every second. To achieve real-time execution, attach this function to a timer.
- Output and update entry-point function, `void rtwdemo_multirate_multitasking_step1(void)`. Call this function periodically at the second fastest rate in the model. For this model, call the function every two seconds. To achieve real-time execution, attach this function to a timer.

Output ports:

- `rtY.Out1` of data type `real_T` with dimension of 1
- `rtY.Out2` of data type `real_T` with dimension of 1

More About

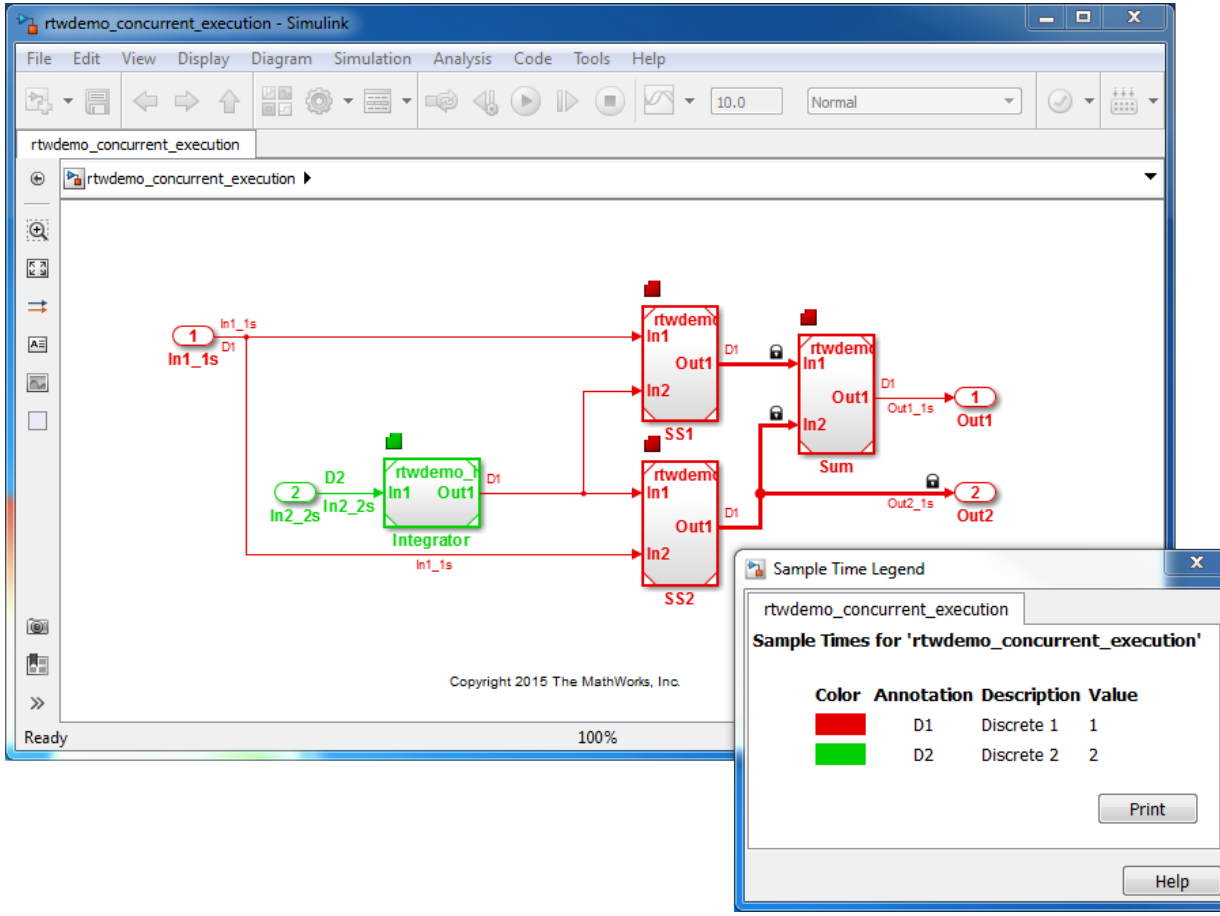
- "Modeling for Multitasking Execution" (Simulink Coder)
- "Deploy Generated Standalone Executable Programs To Target Hardware"
- "Customize Code Organization and Format"

Model Concurrent Execution for Symmetric Multicore CPU Platforms

Use Simulink® time-based, multitask scheduling to simulate and generate code for an application algorithm captured in a single model hierarchy. The model is designed and configured for an embedded system intended to execute on a symmetric multicore, multitasking platform.

Periodic Multirate Model Set Up for Multitasking Concurrent Execution

Open the example model `rtwdemo_concurrent_execution`. The model is configured to display color-coded sample times with annotations. To see them, after opening the model, update the diagram by pressing **Ctrl+D**. To display the legend, press **Ctrl+J**.



Simulink supports simulating concurrent task execution by assigning partitions of a model to tasks that you designate to run concurrently on multicore hardware. Use an implicit or explicit approach to designating partitions.

Simulink implicit partitioning:

- Partitions the model based on sample times specified in the model.
- Assigns a task to each sample rate and designates that the tasks run concurrently.
- Controls the granularity of partitions. For example, you cannot split a sample rate into multiple tasks.

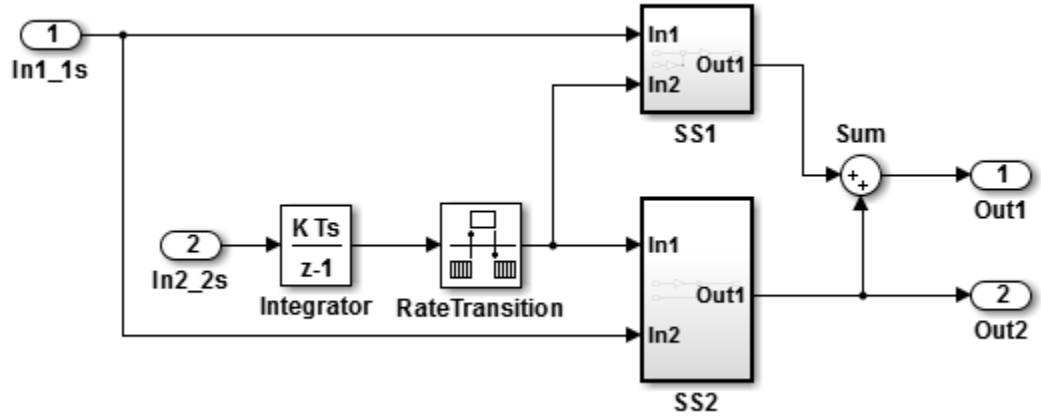
- Does not impose modeling constraints.
- Provides ready-to-use hardware solutions, such as solutions that the Simulink® Real-Time™ product produces.
- Is not relevant to standalone production code generation due to the lack of control over partition granularity.

Explicit partitioning:

- Use Model and Subsystem blocks to partition the model.
- Create an arbitrary number of tasks.
- Simulink assigns each partition to a task.
- Simulink imposes modeling constraints.
- Control the granularity of partitions.
- Split a sample rate into multiple tasks.
- Assign partitions to different processor cores.
- Is for standalone production code generation due to the level of control you have over granularity of partitions.

This example shows explicit partitioning.

Consider the following periodic multirate model that is set up for multitasking execution.



- Sample times for Inport blocks `In1_1s` and `In2_2s` are set to 1 and 2 seconds, respectively.

- To provide a clear partitioning of rates, sample times for models SS1 and SS2 are set to 1.
- The Rate Transition block explicitly models a rate transition.

To support concurrent execution of tasks in a multicore run-time environment, the preceding model was modified:

- The Integrator block is in a Model block configured with a fixed-step discrete solver and a step size of two seconds.
- Subsystems SS1 and SS2 were converted to Model blocks configured with a fixed-step discrete solver and a step size of one second.
- The Sum block is in a Model block configured with a fixed-step discrete solver and a step size of one second. Another option for the Sum block is to place it in SS1 or SS2 and compute its value coincident with the Model block. For concurrent execution of tasks, only connection blocks, Model blocks, and Subsystem blocks can be at the root level of a model.
- The Rate Transition block was removed.

Relevant Model Configuration Parameter Settings

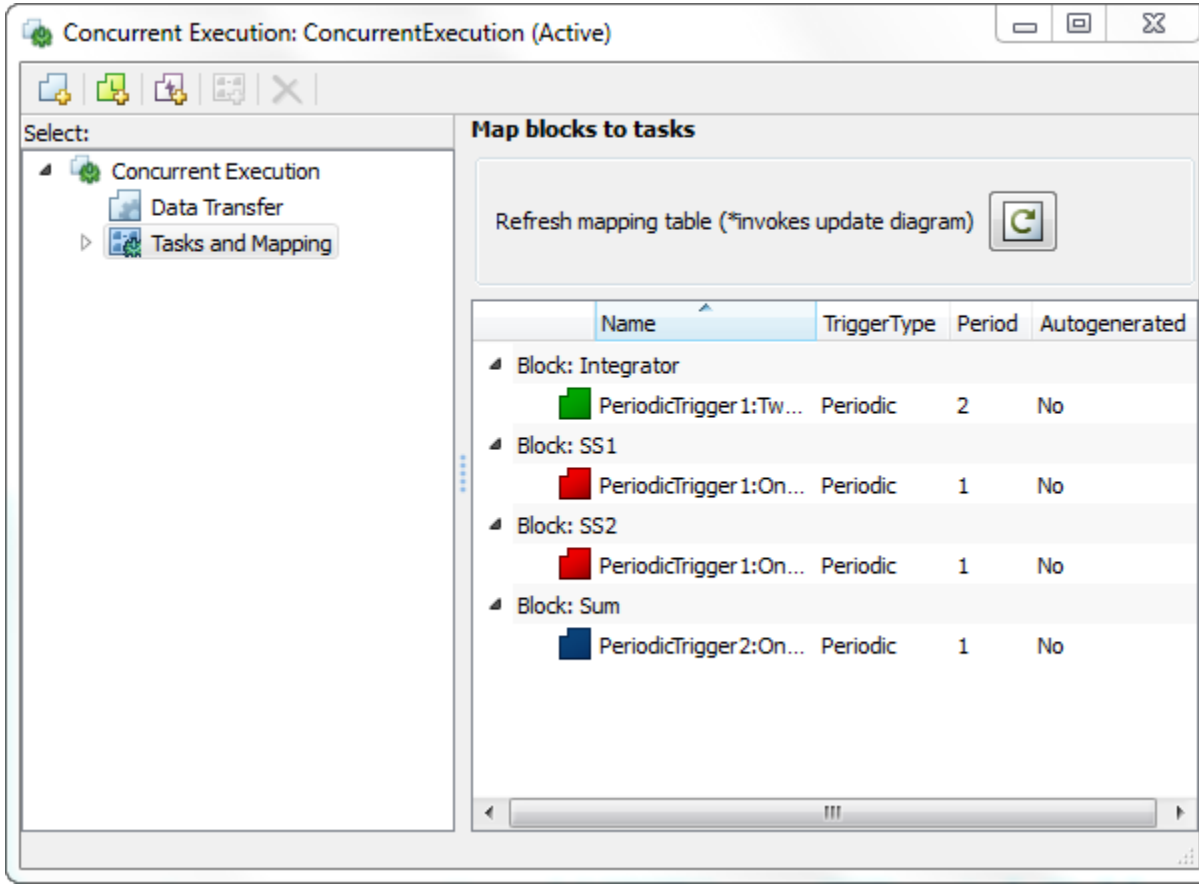
- **Solver > Type** set to `Fixed-step`.
- **Solver > Solver** set to `discrete (no continuous states)`.
- **Solver > Treat each discrete rate as a separate task** selected.
- **Solver > Automatically handle rate transition for data transfer** selected. Necessary because Rate Transition block was removed.
- **Solver > Allow tasks to execute concurrently on target** selected.

Concurrent Execution Parameter Settings

Open the Concurrent Execution dialog box by clicking **Configure Tasks** on the Configuration Parameters **Solver** pane. Selecting **Allow tasks to execute concurrently on target** enables the **Configure Tasks** button.

When selected, the **Enable explicit model partitioning for concurrent behavior** parameter enables concurrent execution options for the top-level model.

Click **Tasks and Mapping** to review the tasks and mapping.



Simulink creates a default mapping for each partition (Model block) by assigning each partition to a separate task. Simulink designates that each partition executes concurrently and simulates latency effects that data communication between processor cores imposes. This dialog box displays a mapping consisting of partitions spread across two independent periodic triggers: `SS1`, `SS2`, and `Sum` mapped to periodic trigger 1 and `Integrator` mapped to periodic trigger 2.

Scheduling

Simulink® simulates the model based on the model configuration. Code generated from the model implements the same execution semantics. Simulink propagates and uses the

Inport block sample times to order block execution based on a multicore, multitasking execution platform.

For this model, the sample time legend shows an implicit rate grouping. Red represents the fastest discrete rate. Green represents the second fastest discrete rate.

The generated code schedules subrates in the model. In this example, the rate for Inport block `In2_2s`, the green rate, is a subrate. The generated code properly transfers data between the rates.

Benefits of implicit Simulink rate grouping:

- Simulink does not impose architectural constraints on the model. Create a model without imposing software architecture constraints within it.
- Your execution framework does not require details about underlying function scheduling and data transfers between rates. Therefore, model interface requirements are simplified. The execution framework uses generated interface code to write input, call the model step function, and read output.
- The code generator optimizes code across rates, based on multitasking execution semantics.

Simulink enforces data transfer constraints:

- Data transfers occur between a single read task and a single write task.
- Tasks run on a single processor.
- Processes do not stop or restart, especially during data transfers between tasks.
- Read and write operations on byte-sized variables are atomic.

Your execution framework can communicate with external devices for reading and writing model input. For example, model external devices by using Simulink S-Function blocks. Generate code for those blocks with the rest of the algorithm.

Generate Code and Report

Generate code and a code generation report. The example model generates a report.

Review Generated Code

From the code generation report, review the generated code.

- `ert_main.c` is an example main program (execution framework) for the model. This code controls model code execution by indirectly

calling entry-point functions `PeriodicTrigger1_OneSecond_step`, `PeriodicTrigger1_TwoSecond_step`, and `PeriodicTrigger2_OneSecond_step` with the function `rtwdemo_concurrent_execution_step`. Use this file as a starting point for coding your execution framework.

- `rtwdemo_concurrent_execution.c` contains entry points for the code that implements the model algorithm. This file includes the rate and task scheduling code.
- `rtwdemo_concurrent_execution.h` declares model data structures and a public interface to the model entry points and data structures.
- `model_reference_types.h` contains type definitions for timing bridges. These type definitions are generated for a model reference target or a model containing Model blocks.
- `rtw_windows.h` declares mutex and semaphore function protoypes that the generated code uses for concurrent execution on Microsoft® Windows® platforms.
- `rtwtypes.h` defines data types, structures, and macros that the generated code requires.

Code Interface

Open and review the Code Interface Report. Use the information in that report to write the interface code for your execution framework:

1. Include the generated header file by adding directive `#include rtwdemo_concurrent_execution.h`.
2. Write input data to the generated code for model Inport blocks.
3. Call the generated entry-point functions.
4. Read data from the generated code for model Outport blocks.

Input ports:

- `In1_1s` of data type `real_T` with dimension of 1
- `In2_2s` of data type `real_T` with dimension of 1

Entry-point functions:

- Initialization entry-point function, `void rtwdemo_concurrent_execution_initialize(void)`. At startup, call this function once.
- Output and update entry-point (step) function, `void PeriodicTrigger1_OneSecond_step(void)`. Call this function periodically for

one of two tasks that require scheduling at the fastest rate in the model. For this model, call the function every second.

- Output and update entry-point function, `void PeriodicTrigger1_TwoSecond_step(void)`. Call this function periodically at the second fastest rate in the model. For this model, call the function every two seconds.
- Output and update entry-point function, `void PeriodicTrigger2_OneSecond_step(void)`. Call this function periodically for the second task that requires scheduling at the fastest rate in the model. For this model, call the function every second.

To achieve real-time execution, define a task or thread for each entry-point step function. Trigger execution of each function based on a timer that has the same rate as the given function. The operating system schedules the tasks across cores dynamically or based on your mapping of tasks to cores.

Output ports:

- `Out1_1s` of data type `real_T` with dimension 1
- `Out2_1s` of data type `real_T` with dimension 1

More About

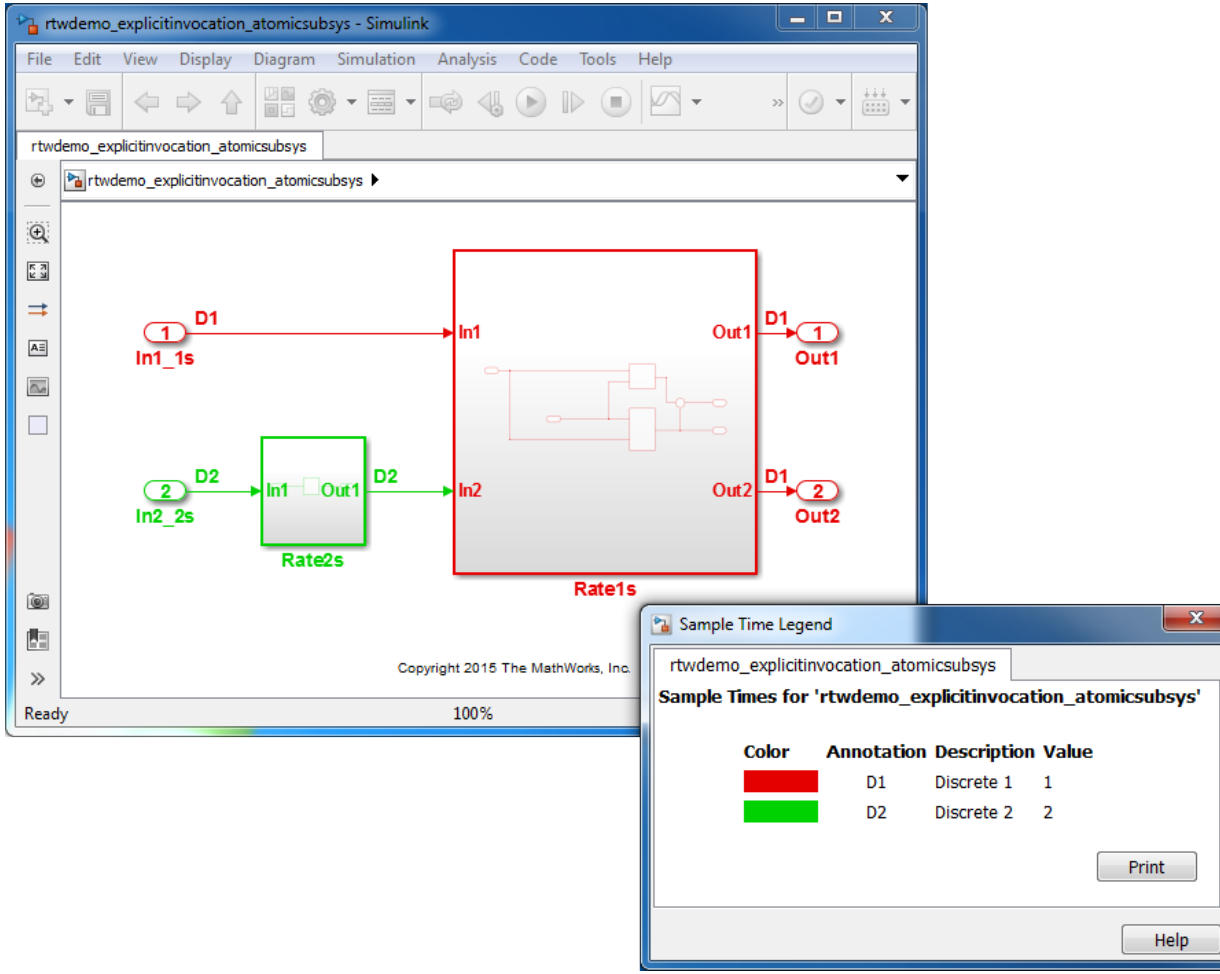
- "Multicore Processor Targets" (Simulink)
- "Deploy Generated Standalone Executable Programs To Target Hardware"
- "Customize Code Organization and Format"

Model Explicit Function Invocation with Atomic Subsystems

Deploy embedded system code from Simulink® models by partitioning a model into multiple atomic subsystems that you build separately.

Atomic Subsystem Model

Open the example model `rtwdemo_explicitinvocation_atomicsubsys`. The model is configured to display color-coded sample times with annotations. To see them, after opening the model, update the diagram by pressing **Ctrl+D**. To display the legend, press **Ctrl+J**.



This model partitions an algorithm into two atomic subsystems: `Rate1s` and `Rate2s`. Subsystem `Rate1s` is configured with a sample time of 1 second. Subsystem `Rate2s` is configured with a sample time of 2 seconds.

Relevant Model Configuration Parameter Settings

- **Solver > Type** set to `Fixed-step`.
- **Solver > Solver** set to `discrete (no continuous states)`.
- **Solver > Treat each discrete rate as a separate task** cleared.

Scheduling

Simulink® simulates the model based on the model configuration. Simulink propagates and uses the Inport block sample times to order block execution based on a single-core, single-tasking execution platform.

For this example, the sample time legend shows implicit rate grouping. Red represents the fastest discrete rate. Green represents the second fastest discrete rate.

Based on ratemonotonic scheduling, your application code (execution framework) must transfer data between subsystems `Rate2s` and `Rate1s` at a frequency of 2 seconds with the priority of 1 second. That is, the generated function transfers data in the 1 second task every other time prior to executing code for subsystem `Rate1s`.

Your execution framework must schedule the generated function code and handle the data transfers between them. This is an advantage for multirate models because the generated code assumes no scheduling or data transfer semantics. However, the execution framework must handle data transfers explicitly.

Generate Code and Report

Generate a single callable function for each subsystem without connections between them. Multiple ways are available to generate code for a subsystem, including from the subsystem context menu. For example, right-click a subsystem block and click **C/C++ Code > Build This Subsystem**. In the Build code for Subsystem dialog box, click **Build**.

The example model generates a report.

Review Generated Code

From the code generation report, review the generated code.

- `ert_main.c` is an example main program (execution framework) for the subsystem. This code controls model code execution by calling entry-point function `Rate1s_step` or `Rate2s_step`. Use this file as a starting point for coding your execution framework.
- `Rate1s.c` and `Rate2s.c` contain entry points for the code that implements subsystem `Rate1s` and `Rate2s`, respectively. This file includes the rate and task scheduling code.
- `Rate1s.h` and `Rate2s.h` declare model data structures and a public interface to subsystem entry points and data structures.

- `rtwtypes.h` defines data types, structures, and macros that the generated code requires.

Code Interface

Open and review the Code Interface Report. Use the information in that report to write the interface code for your execution framework:

1. Include the generated header file by adding directive `#include rtwdemo_explicitinvocation_atomicsusys.h`.
2. Write input data to the generated code for model Inport blocks.
3. Call the generated entry-point functions.
4. Read data from the generated code for model Outport blocks.

Input ports, `Rate1s`:

- `rtU.In1` of type `real_T` with dimension of 1
- `rtU.In2` of type `real_T` with dimension of 1

Entry-point functions, `Rate1s`:

- Initialize entry-point function, `void Rate1s_initialize(void)`. At startup, call this function once.
- Output and update entry-point (step) function, `void Rate1s_step(void)`. Call this function periodically, every second.
- Termination function, `void Rate1s_terminate(void)`. Call this function once from your shutdown code.

Output ports, `Rate1s`:

- `rtY.Out1` of type `real_T` with dimension of 1
- `rtY.Out2` of type `real_T` with dimension of 1

Input ports, `Rate2s`:

- `rtU.In1` of type `real_T` with dimension of 1

Entry-point functions, `Rate2s`:

- Initialize entry-point function, `void Rate2s_initialize(void)`. Call this function once at startup.

- Output and update entry-point (step), `void Rate2s_step(void)`. Call this function periodically, every 2 seconds.
- Termination function, `void Rate2s_terminate(void)`. Call this function once from your shutdown code.

Output ports, `Rate2s`:

- `rtY.Out1` of type `real_T` with dimension of 1

More About

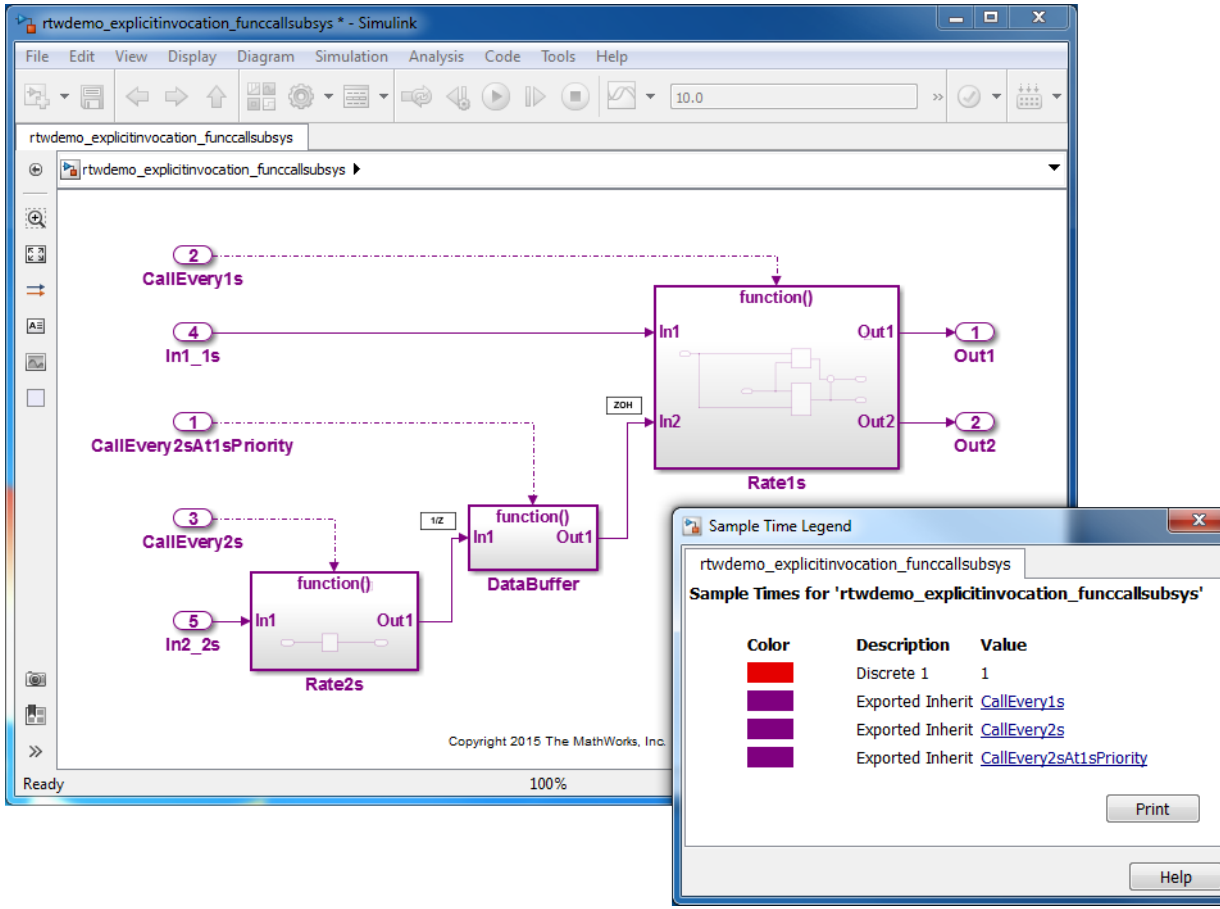
- "Generate Modular Function Code"
- "Deploy Generated Standalone Executable Programs To Target Hardware"
- "Customize Code Organization and Format"

Model Explicit Function Invocation with Function-Call Subsystems

Deploy embedded system code from Simulink® models by partitioning a model into function-call subsystems that you build separately.

Function-Call Subsystem Model

Open the example model `rtwdemo_explicitinvocation_funccallsubsys`. The model is configured to display color-coded sample times with annotations. To see them, after opening the model, update the diagram by pressing **Ctrl+D**. To display the legend, press **Ctrl+J**.



This model partitions an algorithm into three function-call subsystems: `Rate1s`, `Rate2s`, and `DataBuffer`. Use function-call subsystems to model multirate systems explicitly.

Subsystems `Rate1s` and `DataBuffer` use a sample time of 1 second. Subsystem `Rate2s` usea a sample time of 2 seconds.

This model design is referred to as export function modeling. Simulink constrains the model to function-call subsystems at the root level. The driving Inport block specifies the function name.

Relevant Model Configuration Parameter Settings

- **Solver > Type** set to `Fixed-step`.
- **Solver > Solver** set to `discrete (no continuous states)`.
- **Solver > Treat each discrete rate as a separate task** selected. Simulink applies multitasking execution because the model uses multiple sample rates.

Scheduling

Simulink® simulates the model based on the model configuration. Simulink propagates and uses the Inport block sample times to order block execution based on a single-core, multitasking execution platform.

In the sample time legend, red identifies the fastest discrete rate. Magenta identifies rates inherited from exported functions, indicating their execution is outside the context of Simulink scheduling.

Your execution framework must schedule the generated function code and transfer data between functions.

Your execution framework needs to schedule the generated function code and handle data transfers between functions. The generated code is simple and you control the order of execution.

Generate Code and Report

Generate code and a code generation report. The example model generates a report.

Review Generated Code

From the code generation report, review the generated code.

- `ert_main.c` is an example main program (execution framework) for the model. This code shows how to call the exported functions. The code also shows how to initialize, execute, and terminate the generated code.
- `rtwdemo_explicitinvocation_funccallsubsys.c` calls the initialization function and exported functions for subsystems `Rate1s`, `Rate2s`, and `DataBuffer`.
- `rtwdemo_explicitinvocation_funccallsubsys.h` declares model data structures and a public interface to the exported entry point functions and data structures.
- `rtwtypes.h` defines data types, structures, and macros that the generated code requires.

Code Interface

Open and review the Code Interface Report. Use the information in that report to write the interface code for your execution framework:

1. Include the generated header files by adding directives `#include Rate1s.h`, `#include DataBuffer.h`, and `#include Rate2s.h`.
2. Write input data to the generated code for model Inport blocks.
3. Call the generated entry-point functions.
4. Read data from the generated code for model Outport blocks.

Input ports:

- `rtU.In1_1s` of type `real_T` with dimension of 1
- `rtU.In2_2s` of type `real_T` with dimension of 1

Entry-point functions:

- Initialize entry-point function, `void rtwdemo_explicitinvocation_funccallsubsys_initialize(void)`. At startup, call this function once.
- Exported function, `void CallEvery1s(void)`. Call this function as needed.
- Exported function, `void CallEvery1s(void)`. Call this function as needed.
- Exported function, `void CallEvery2sAt1sPriority(void)`. Call this function as needed.

Output ports:

- `rtY.Out1` of type `real_T` with dimension of 1
- `rtY.Out2` of type `real_T` with dimension of 1

More About

- "Generate Component Source Code for Export to External Code Base"
- "Deploy Generated Standalone Executable Programs To Target Hardware"
- "Customize Code Organization and Format"

Model for AUTOSAR Platform

Different ways are available for using Simulink® to model AUTOSAR atomic software components.

Prerequisites

The Embedded Coder Support Package for AUTOSAR Standard is required for generating C and ARXML code for AUTOSAR atomic software components.

Install the AUTOSAR Standard Support Package

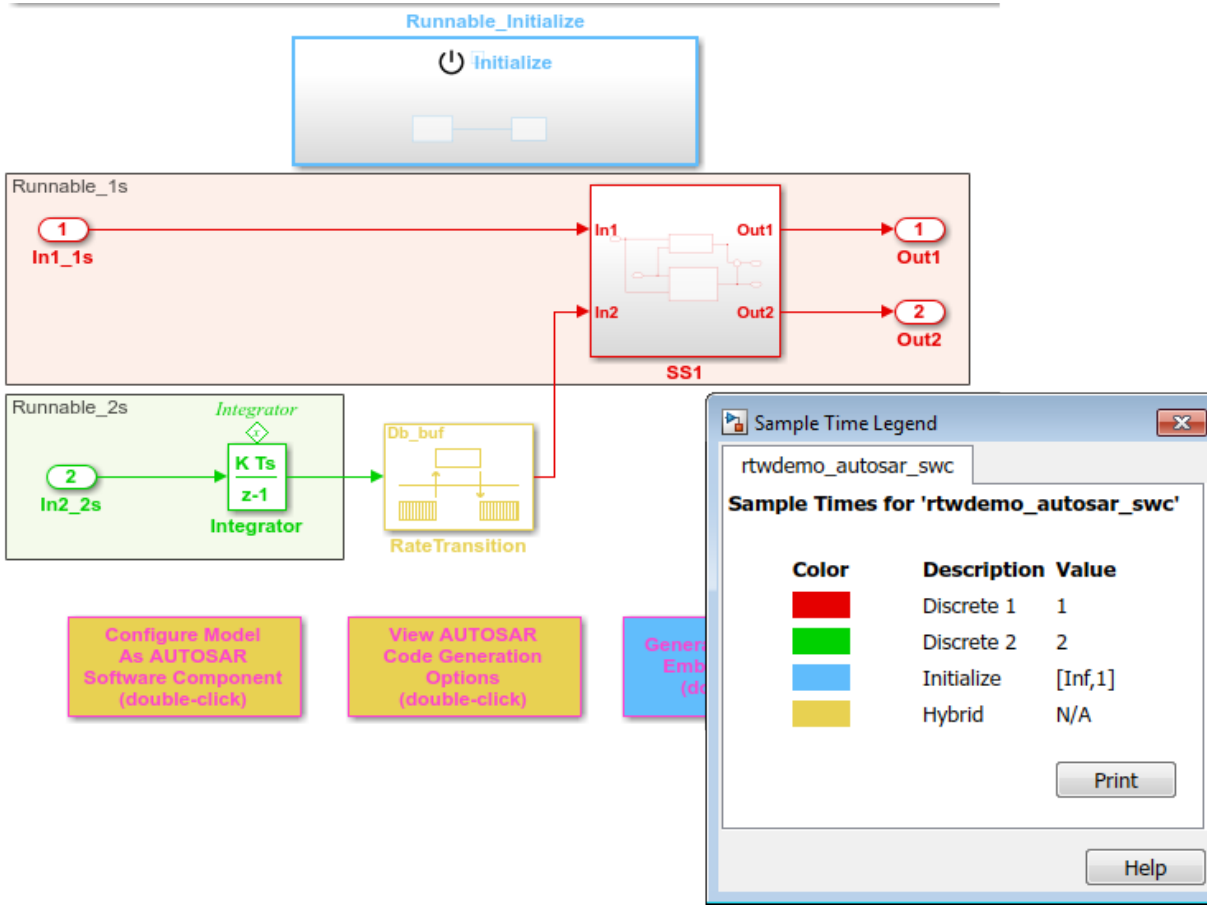
"Support Package Installation" (MATLAB)

Multiple Periodic Runnables Configured for Multitasking

Open the example model `rtwdemo_autosar_swc`. The model shows the implementation of an AUTOSAR atomic software component (ASWC). Two periodic runnables, `Runnable_1s` and `Runnable_2s`, are modeled with multiple sample rates: 1 second (`In1_1s`) and 2 seconds (`In2_2s`). To maximize execution efficiency, the model is configured for multitasking.

The model includes an Initialize Function block, which initializes the integrator in `Runnable_2s` to a value of 1.

To display color-coded sample rates with annotations and a legend, select **Display > Sample Time > Colors**.



Relevant Model Configuration Parameter Settings

- **Solver > Type** set to `Fixed-step`.
- **Solver > Solver** set to `discrete (no continuous states)`.
- **Solver > Fixed-step size (fundamental sample time)** set to `auto`.
- **Solver > Treat each discrete rate as a separate task** selected.

Scheduling

In the model window, enable sample time color-coding by selecting **Display > Sample Time > Colors**. The sample time legend shows the implicit rate grouping. Red

represents the fastest discrete rate. Green represents the second fastest discrete rate. Yellow represents the mixture of the two rates.

Because the model has multiple rates and the **Solver** parameter **Treat each discrete rate as a separate task** is selected, the model simulates in multitasking mode. The model handles the rate transition for `In2_2s` explicitly with the Rate Transition block.

The Rate Transition block parameter **Ensure deterministic data transfer** is cleared to facilitate integration into an AUTOSAR RTE.

The generated code for the model schedules subrates in the model. In this example, the rate for Inport block `In2_2s`, the green rate, is a subrate. The generated code properly transfers data between tasks that run at the different rates.

Generate Code and Report

Generate code and a code generation report. The example model generates a report.

Generated code complies with AUTOSAR so that you can schedule the code with the AUTOSAR run-time environment (RTE).

Review Generated Code

In the code generation report, review the generated code.

- `rtwdemo_autosar_swc.c` contains entry points for the code that implements the model algorithm. This file includes the rate scheduling code.
- `rtwdemo_autosar_swc.h` declares model data structures and a public interface to the model entry points and data structures.
- `rtwdemo_autosar_swc_private.h` contains local `define` constants and local data required by the model and subsystems.
- `rtwdemo_autosar_swc_types.h` provides forward declarations for the real-time model data structure and the parameters data structure.
- `rtwtypes.h` defines data types, structures, and macros that the generated code requires.
- `rtwdemo_autosar_swc_component.arxml`, `rtwdemo_autosar_swc_datatype.arxml`, `rtwdemo_autosar_swc_implementation.arxml`, and `rtwdemo_autosar_swc_interface.arxml` contain elements and objects that represent AUTOSAR software components, ports, interfaces, data types, and

packages. You import these files into the Simulink environment by using the AUTOSAR `arxml` importer tool.

- `Compiler.h`, `Platform_Types.h`, `Rte_ASWC.h`, `Rte_Type.h`, and `Std_Types.h` contain stub implementations of AUTOSAR RTE functions. Use these files to test the generated code in Simulink, for example, in software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations of the component under test.

Code Interface

Open and review the Code Interface Report. This information is captured in the ARXML files. The RTE generator uses the ARXML to interface the code into an AUTOSAR RTE.

Input ports:

- Require port, interface: sender-receiver of type `real-T` of 1 dimension
- Require port, interface: sender-receiver of type `real-T` of 1 dimension

Entry-point functions:

- Initialization entry-point function, `void Runnable_Initialize(void)`. At startup, call this function once.
- Output and update entry-point function, `void Runnable_1s(void)`. Call this function periodically at the fastest rate in the model. For this model, call the function every second. To achieve real-time execution, attach this function to a timer.
- Output and update entry-point function, `void Runnable_2s(void)`. Call this function periodically at the second fastest rate in the model. For this model, call the function every 2 seconds. To achieve real-time execution, attach this function to a timer.

Output ports:

- Provide port, interface: sender-receiver of type `real-T` of 1 dimension
- Provide port, interface: sender-receiver of type `real-T` of 1 dimension

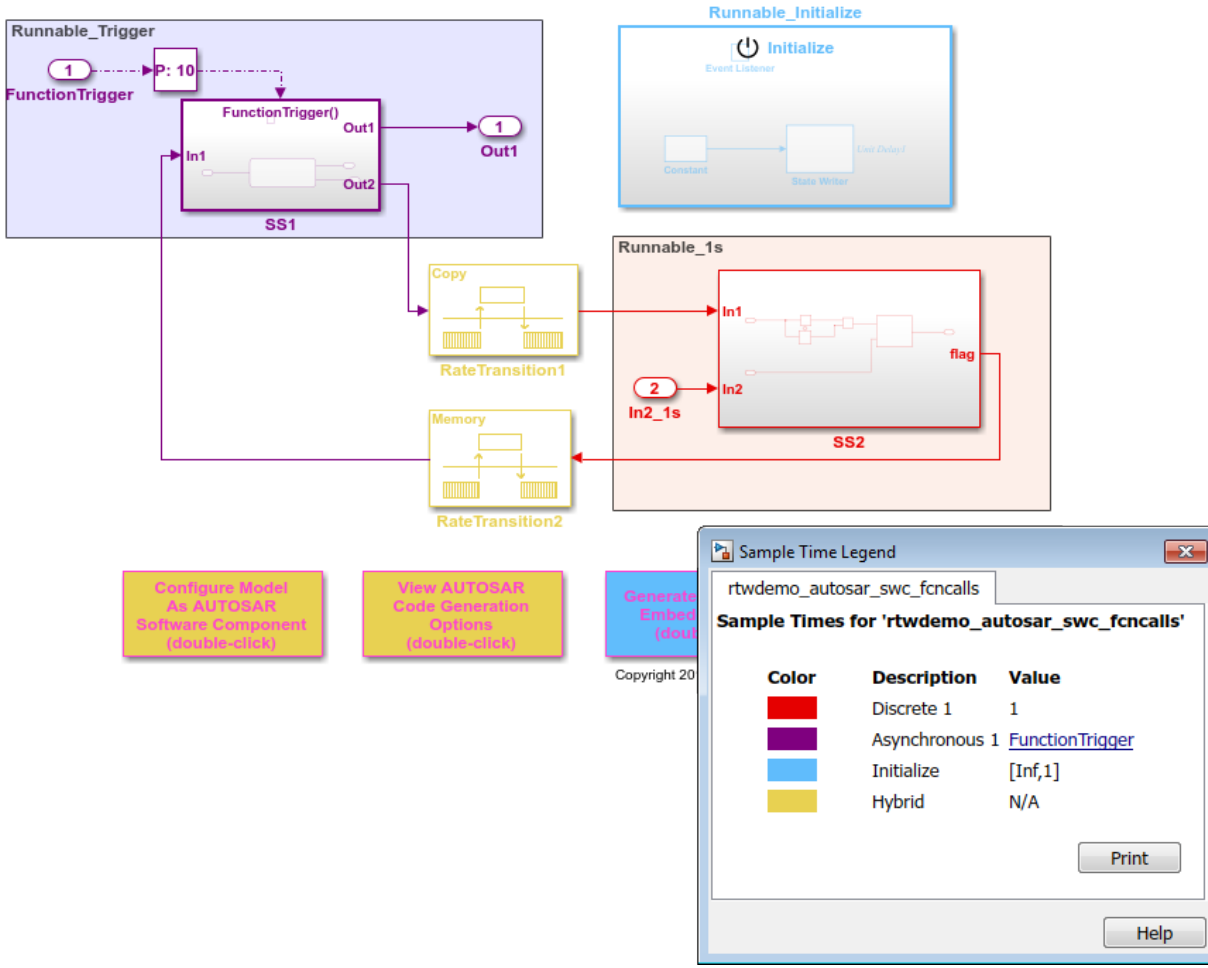
Multiple Runnables Configured as Periodic-Rate Runnable and Asynchronous Function-Call Runnable

Open the example model `rtwdemo_autosar_swc_fcncalls`. The model shows the implementation of an AUTOSAR atomic software component (ASWC). The model uses an asynchronous function-call runnable, `Runnable_Trigger`, which is triggerd by an external event. The model also includes a periodic rate-based runnable, `Runnable_1s`. The Rate Transition blocks represent interrunnable variables.

Use this approach to model the JMAAB complex control model type beta architecture. In JMAAB type beta modeling, at the top level of a control model, you place function layers above scheduling layers.

The model includes an Initialize Function block, which initializes the unit delay in `Runnable_1s` to a value of 0.

To display color-coded sample rates with annotations and a legend, select **Display > Sample Time > Colors**.



Relevant Model Configuration Parameter Settings

- **Solver > Type** set to `Fixed-step`.
- **Solver > Solver** set to `discrete (no continuous states)`.
- **Solver > Fixed-step size (fundamental sample time)** set to 1.
- **Solver > Treat each discrete rate as a separate task** cleared.

Scheduling

In the model window, enable sample time color-coding by selecting **Display > Sample Time > Colors**. The sample time legend shows the implicit rate grouping. Red represents the discrete rate. Magenta represents the asynchronous function trigger. Yellow represents the mixture of two rates.

The asynchronous trigger runnable runs at asynchronous rates (the **Sample time type** parameter of the function-call subsystem Trigger block is set to |triggered]) while the periodic rate runnable runs at the specified discrete rate. The generated code manages the rates by using single-tasking assumptions. For models with one discrete rate, the code generator does not produce scheduling code because there is only a single rate to execute. Use this technique for a single-rate application when you have one periodic runnable.

The model handles transitions between the asynchronous and discrete rates of the interrunnables with the two Rate Transition blocks. The Rate Transition block parameter **Ensure deterministic data transfer** is cleared to facilitate integration into an AUTOSAR RTE.

Generate Code and Report

Generate code and a code generation report. The example model generates a report.

Generated code complies with AUTOSAR so that you can schedule the code with the AUTOSAR run-time environment (RTE).

Review Generated Code

In the code generation report, review the generated code.

- `rtwdemo_autosar_swc_fcncalls.c` contains entry points for the code that implements the model algorithm. This file includes the rate scheduling code.

- `rtwdemo_autosar_swc_fcncalls.h` declares model data structures and a public interface to the model entry points and data structures.
- `rtwdemo_autosar_swc_fcncalls_private.h` contains local `define` constants and local data required by the model and subsystems.
- `rtwdemo_autosar_swc_fcncalls_types.h` provides forward declarations for the real-time model data structure and the parameters data structure.
- `rtwtypes.h` defines data types, structures, and macros that the generated code requires.
- `rtwdemo_autosar_swc_fcncalls_component.arxml`, `rtwdemo_autosar_swc_fcncalls_datatype.arxml`, `rtwdemo_autosar_swc_fcncalls_implementation.arxml`, and `rtwdemo_autosar_swc_fcncalls_interface.arxml` contain elements and objects that represent AUTOSAR software components, ports, interfaces, data types, and packages. You import these files into the Simulink environment by using the AUTOSAR `arxml` importer tool.
- `Compiler.h`, `Platform_Types.h`, `Rte_ASWC.h`, `Rte_Type.h`, and `Std_Types.h` contain stub implementations of AUTOSAR RTE functions. Use these files to test the generated code in Simulink, for example, in software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations of the component under test.

Code Interface

Open and review the Code Interface Report. This information is captured in the ARXML files. The RTE generator uses the ARXML to interface the code into an AUTOSAR RTE.

Input port:

- Require port, interface: sender-receiver of type `real-T` of 1 dimension

Entry-point functions:

- Initialization entry-point function, `void Runnable_Initialize(void)`. At startup, call this function once.
- Simulink function, `void Runnable_1s(void)`. Call this function periodically at the fastest rate in the model. For this model, call the function every second. To achieve real-time execution, attach this function to a timer.
- Exported function, `void Runnable_Trigger(void)`. Call this function at any time from an external trigger.

Output port:

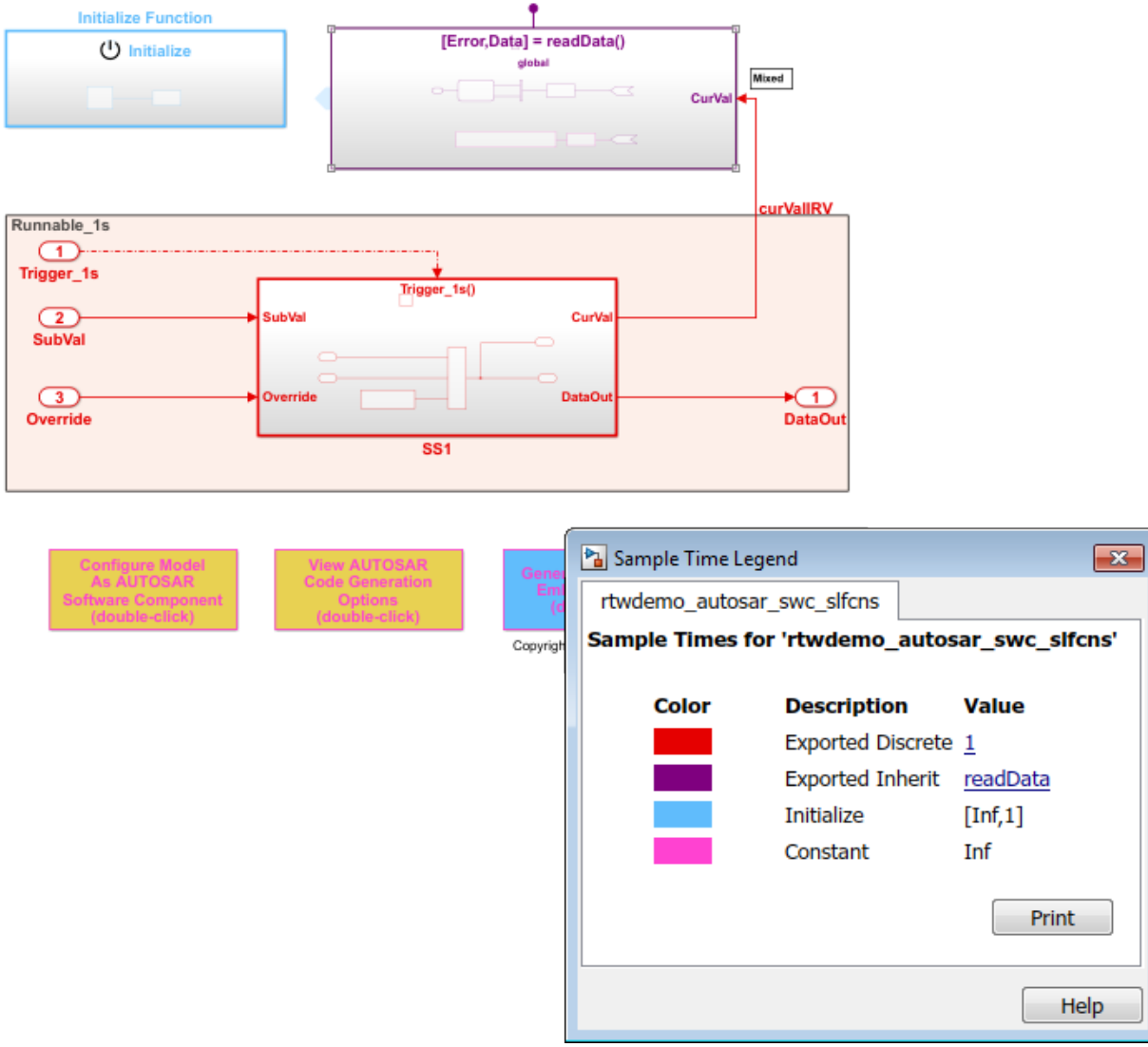
- Provide port, interface: sender-receiver of type `real-T` of 1 dimension

Multiple Runnables Configured As Function-Call Subsystem and Simulink Function

Open the example model `rtwdemo_autosar_swc_slfcns`. The model shows the implementation of an AUTOSAR atomic software component (ASWC). The model includes one periodic rate runnable, `Runnable_1s`, that uses a function-call subsystem, `SS1`. The model also includes a Simulink function, `readData`, to provide a value (`CurVal`) to clients that request it.

The model includes an Initialize Function block, which initializes the unit delay in subsystem `RollingCounter` to a value of 0.

To display color-coded sample rates with annotations and a legend, select **Display > Sample Time > Colors**.



Use function-call subsystems:

- When it is difficult or not possible to specify system events in a Simulink model.

- To achieve complex multirate scheduling of runnables. Model each rate as a separate function-call subsystem.

Relevant Model Configuration Parameter Settings

- **Solver > Type** set to `Fixed-step`.
- **Solver > Solver** set to `discrete (no continuous states)`.
- **Solver > Fixed-step size (fundamental sample time)** set to 1.
- **Solver > Treat each discrete rate as a separate task** selected.

Scheduling

In the model window, enable sample time color-coding by clicking **Display > Sample Time > Colors**. The sample time legend shows the implicit rate grouping. Red identifies the discrete rate. Magenta identifies rates inherited from exported functions, indicating their execution is outside the context of Simulink scheduling.

Your execution framework must schedule the generated function code and handle data transfers between functions.

Generate Code and Report

Generate code and a code generation report. The example model generates a report.

The code generator:

- Produces an AUTOSAR runnable for the function-call subsystem at the root level of the model.
- Implements signal connections between runnables as AUTOSAR interrunable variables (IRVs).

Generated code complies with AUTOSAR so that you can schedule the code with the AUTOSAR run-time environment (RTE).

Review Generated Code

In the code generation report, review the generated code.

- `rtwdemo_autosar_swc_slfcns.c` contains entry points for the code that implements the model algorithm. This file includes the rate scheduling code.
- `rtwdemo_autosar_swc_slfcns.h` declares model data structures and a public interface to the model entry points and data structures.

- `rtwdemo_autosar_swc_slfcns_private.h` contains local `define` constants and local data required by the model and subsystems.
- `rtwdemo_autosar_swc_slfcns_types.h` provides forward declarations for the real-time model data structure and the parameters data structure.
- `readData.c` contains code for the Simulink function.
- `readData_private.h` contains local `define` constants and local data required by the function.
- `readData.h` declares data structures and a public interface for calling the function.
- `rtwtypes.h` defines data types, structures, and macros that the generated code requires.
- `rtwdemo_autosar_swc_slfcns_component.arxml`, `rtwdemo_autosar_swc_slfcns_datatype.arxml`, `rtwdemo_autosar_swc_slfcns_implementation.arxml`, and `rtwdemo_autosar_swc_slfcns_interface.arxml` contain elements and objects that represent AUTOSAR software components, ports, interfaces, data types, and packages. You import these files into the Simulink environment by using the AUTOSAR `arxml` importer tool.
- `Compiler.h`, `Platform_Types.h`, `Rte_ASWC.h`, `Rte_Type.h`, and `Std_Types.h` contain stub implementations of AUTOSAR RTE functions. Use these files to test the generated code in Simulink, for example, in software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations of the component under test.

Code Interface

Open and review the Code Interface Report. This information is captured in the ARXML files. The RTE generator uses the ARXML to interface the code into an AUTOSAR RTE.

Input ports:

- Require port, interface: sender-receiver of type `uint16-T` of 1 dimension
- Require port, interface: sender-receiver of type `real-T` of 1 dimension Entry-point functions:

Entry-point functions:

- Initialization entry-point function, `void Runnable_Init(void)`. At startup, call this function once.
- Exported function, `void Runnable_1s(void)`. Call this function periodically, every second.

- Simulink function, `Std_ReturnType readData(real_T Data[2])`. Call this function at any time.

Output ports:

- Provide port, interface: sender-receiver of type `uint16-T` of 1 dimension

More About

- "AUTOSAR Component Creation"
- "AUTOSAR Code Generation"

Modeling in Simulink Coder

Configure a Model for Code Generation

Model configuration parameters determine the method for generating the code and the resulting format.

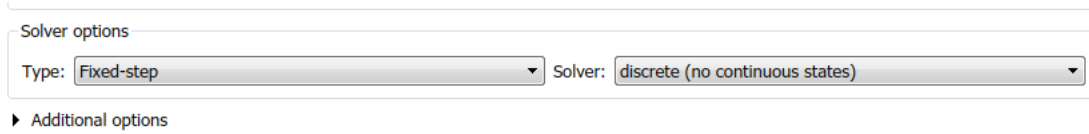
1 Open `rtwdemo_throttlecntrl` and save a copy as `throttlecntrl` in a writable location on your MATLAB path.

Note: This model uses Stateflow® software.

---

2 Open the Configuration Parameters dialog box **Solver** pane. To generate code for a model, you must configure the model to use a fixed-step solver. For this example, set the parameters as noted in the following table.

Parameter	Setting	Effect on Generated Code
Type	`Fixed-step`	Maintains a constant (fixed) step size, which is required for code generation
Solver	`discrete (no continuous states)`	Applies a fixed-step integration technique for computing the state derivative of the model
Fixed-step size	`.001`	Sets the base rate; must be the lowest common multiple of all rates in the system



3 Open the **Code Generation** pane and make sure that **System target file** is set to `grt.tlc`.

Note: The GRT (Generic Real-Time Target) configuration requires a fixed-step solver. However, the `rsim.tlc` system target file supports variable step code generation.

The system target file (STF) defines a target, which is an environment for generating and building code for execution on a certain hardware or operating system platform. For example, one property of a target is code format. The grt configuration requires a fixed step solver and the rsim.tlc supports variable step code generation.

4 Open the **Code Generation** > **Custom Code** pane, and under **Include list of additional**, select **Include directories**. In the **Include directories** text field, enter:

```
"$matlabroot$\toolbox\rtw\rtwdemos\EmbeddedCoderOverview\"
```

This directory includes files that are required to build an executable for the model.

5 Apply your changes and close the dialog box.

Supported Products and Block Usage

In this section...

Related Products

The following table summarizes MathWorks products that extend and complement Simulink Coder software. For information about these and other MathWorks products, see `www.mathworks.com`.

Product	Extends Code Generation Capabilities for ...
Aerospace Blockset™	Aircraft, spacecraft, rocket, propulsion systems, and unmanned airborne vehicles
Audio System Toolbox™	Audio processing systems
Automated Driving System Toolbox™	Designing, simulating, and testing ADAS and autonomous driving systems
Communications System Toolbox™	Physical layer of communication systems
Computer Vision System Toolbox™	Video processing, image processing, and computer vision systems
Control System Toolbox™	Linear control systems
DSP System Toolbox™	Signal processing systems
Embedded Coder	Embedded systems, rapid prototyping boards, and microprocessors in mass production
Fixed-Point Designer™	Fixed-point systems
Fuzzy Logic Toolbox™	System designs based on fuzzy logic
HDL Verifier™	Direct programming interface (DPI) component and transaction-level model (TLM) generation from Simulink
IEC Certification Kit	ISO 26262 and IEC 61508 certification

Product	Extends Code Generation Capabilities for ...
Model-Based Calibration Toolbox™	Developing processes for systematically identifying optimal balance of engine performance, emissions, and fuel economy, and reusing statistical models for control design, hardware-in-the-loop (HIL) testing, or powertrain simulation
Model Predictive Control Toolbox™	Controllers that optimize performance of multi-input and multi-output systems that are subject to input and output constraints
Neural Network Toolbox™	Neural networks
Phased Array System Toolbox™	Sensor array systems in radar, sonar, wireless communications, and medical imaging applications
Polyspace® Bug Finder™	MISRA-C compliance and static analysis of generated code
Polyspace Code Prover™	Formal analysis of generated code
Powertrain Blockset™	Real-time testing of powertrain applications
Robotics System Toolbox™	Robot Operating System (ROS) node generation
Simscape™	Systems spanning mechanical, electrical, hydraulic, and other physical domains as physical networks
Simscape Driveline™	Driveline (drivetrain) systems
Simscape Electronics™	Electronic and electromechanical systems
Simscape Fluids™	Hydraulic power and control systems
Simscape Multibody™	Three-dimensional mechanical systems
Simscape Power Systems™	Systems that generate, transmit, distribute, and consume electrical power
Simulink 3D Animation™	Systems with 3D visualizations
Simulink Code Inspector™	Automated reviews of generated code
Simulink Design Optimization™	Systems requiring maximum overall system performance

Product	Extends Code Generation Capabilities for ...
Simulink Desktop Real-Time™	Rapid prototyping or hardware-in-the-loop (HIL) simulation of control system and signal processing algorithms
Simulink Real-Time™	Rapid control prototyping, hardware-in-the-loop (HIL) simulation, and other real-time testing applications
Simulink Report Generator™	Automatically generating project documentation in a standard format
Simulink Test™	Software-in-the-loop (SIL), processor-in-the-loop (PIL), and real-time hardware-in-the-loop (HIL) testing of generated code
Simulink Verification and Validation™	Applications requiring automated requirements tracing, model standards compliance checking, and test harness generation
Stateflow	State machines and flow charts
System Identification Toolbox™	Systems constructed from measured input-output data Support exceptions: • Nonlinear IDNLGREY Model, IDDATA Source, IDDATA Sink, and estimator blocks • Nonlinear ARX models that contain custom regressors • `neuralnet` nonlinearities • `customnet` nonlinearities
Vehicle Network Toolbox™	CAN blocks for Accelerator and Rapid Accelerator simulations and code deployment on Windows®

Simulink Built-In Blocks That Support Code Generation

The following tables summarize code generator support for Simulink blocks. There is a table for each block library. For more detail, including data types each block supports, in

the MATLAB Command Window, type `showblockdatatypetable`, or consult the block reference pages. For some blocks, the generated code might rely on `memcpy` or `memset` (`string.h`).

- Additional Math and Discrete: Additional Discrete
- Additional Math and Discrete: Increment/Decrement
- Continuous
- Discontinuities
- Discrete
- Logic and Bit Operations
- Lookup Tables
- Math Operations
- Model Verification
- Model-Wide Utilities
- Ports & Subsystems
- Signal Attributes
- Signal Routing
- Sinks
- Sources
- User-Defined

Additional Math and Discrete: Additional Discrete

Block	Support Notes
Fixed-Point State-Space	The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the **Treat as atomic unit** option.
Transfer Fcn Direct Form II	
Transfer Fcn Direct Form II Time Varying	

Additional Math and Discrete: Increment/Decrement

Block	Support Notes
Decrement Real World	The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the **Treat as atomic unit** option.
Decrement Stored Integer	
Decrement Time To Zero	Supports code generation.
Decrement To Zero	The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the **Treat as atomic unit** option.
Increment Real World	
Increment Stored Integer	

Continuous

Block	Support Notes
Derivative	Not recommended for production-quality code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. The code generated can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. In general, consider using the Simulink Model Discretizer to map continuous blocks into discrete equivalents that support production code generation. To start the Model Discretizer, select **Analysis > Control Design > Model Discretizer**. One exception is the Second-Order Integrator block because, for this block, the Model Discretizer produces an approximate discretization.
Integrator	
Integrator Limited (Simulink)	
PID Controller	
PID Controller (2DOF)	
Second-Order Integrator (Simulink)	
Second-Order Integrator Limited (Simulink)	
State-Space	
Transfer Fcn	
Transport Delay	
Variable Time Delay (Simulink)	

Block	Support Notes
Variable Transport Delay	
Zero-Pole	

Discontinuities

Block	Support Notes
Backlash	Supports code generation.
Coulomb and Viscous Friction (Simulink)	The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the **Treat as atomic unit** option.
Dead Zone	Supports code generation.
Dead Zone Dynamic	The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the **Treat as atomic unit** option.
Hit Crossing	Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.
Quantizer	Supports code generation.
Rate Limiter	Cannot use inside a triggered subsystem hierarchy.
Rate Limiter Dynamic	The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked

Block	Support Notes
	subsystem block to execute as an atomic unit by selecting the **Treat as atomic unit** option.
Relay	Support code generation.
Saturation	
Saturation Dynamic	The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the **Treat as atomic unit** option.
Wrap To Zero	

Discrete

Block	Support Notes
Delay	Supports code generation.
Difference	• The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the **Treat as atomic unit** option. • Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.
Discrete Derivative	• Depends on absolute time when used inside a triggered subsystem hierarchy. • Supports code generation.
Discrete Filter	Support code generation.
Discrete FIR Filter	

Block	Support Notes
PID Controller	• Depends on absolute time when used inside a triggered subsystem hierarchy. • Support code generation.
PID Controller (2DOF)	
Discrete State-Space	Support code generation.
Discrete Transfer Fcn	
Discrete Zero-Pole	
Discrete-Time Integrator	Depends on absolute time when used inside a triggered subsystem hierarchy.
Enabled Delay (Simulink)	Supports code generation.
First-Order Hold	Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.
Memory	Support code generation.
Resettable Delay	
Tapped Delay	
Transfer Fcn First Order	The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the **Treat as atomic unit** option.
Transfer Fcn Lead or Lag	
Transfer Fcn Real Zero	
Unit Delay	Support code generation.
Variable Integer Delay	
Zero-Order Hold	

Logic and Bit Operations

Block	Support Notes
Bit Clear	Support code generation.
Bit Set	
Bitwise Operator	
Combinatorial Logic	
Compare to Constant	
Compare to Zero	
Detect Change	
Detect Decrease	
Detect Fall Negative	
Detect Fall Nonpositive	
Detect Increase	
Detect Rise Nonnegative	
Detect Rise Positive	
Extract Bits	
Interval Test	
Interval Test Dynamic	
Logical Operator	
Relational Operator	
Shift Arithmetic	

Lookup Tables

Block	Support Notes
Cosine	The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the **Treat as atomic unit** check box.
Direct Lookup Table (n-D)	Support code generation.

Block	Support Notes
Interpolation Using Prelookup	
1-D Lookup Table	
2-D Lookup Table	
n-D Lookup Table	
Lookup Table Dynamic	
Prelookup	
Sine (Simulink)	The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the **Treat as atomic unit** option.

Math Operations

Block	Support Notes
Abs	Support code generation.
Add	
Algebraic Constraint	Ignored during code generation.
Assignment	Support code generation.
Bias	
Complex to Magnitude-Angle	
Complex to Real-Imag	
Divide	
Dot Product	
Find Nonzero Elements (Simulink)	
Gain	
Magnitude-Angle to Complex	

Block	Support Notes
Math Function (10^u)	
Math Function (conj)	
Math Function (exp)	
Math Function (hermitian)	
Math Function (hypot)	
Math Function (log)	
Math Function (log10)	
Math Function (magnitude^2)	
Math Function (mod)	
Math Function (pow)	
Math Function (reciprocal)	
Math Function (rem)	
Math Function (square)	
Math Function (transpose)	
Matrix Concatenate (Simulink)	
MinMax	
MinMax Running Resettable	
Permute Dimensions	
Polynomial	
Product	
Product of Elements	
Real-Imag to Complex	
Reciprocal Sqrt (Simulink)	
Reshape	
Rounding Function	
Sign	

Block	Support Notes
Signed Sqrt (Simulink)	
Sine Wave Function	• Does not refer to absolute time when configured for sample-based operation. Depends on absolute time when in time-based operation. • Depends on absolute time when used inside a triggered subsystem hierarchy.
Slider Gain	Support code generation.
Sqrt	
Squeeze	
Subtract	
Sum	
Sum of Elements	
Trigonometric Function	Functions `asinh`, `acosh`, and `atanh` are not supported by all compilers. If you use a compiler that does not support those functions, the software issues a warning for the block and the generated code fails to link.
Unary Minus	Support code generation.
Vector Concatenate (Simulink)	
Weighted Sample Time Math	

Model Verification

Block	Support Notes
Assertion	Supports code generation.
Check Discrete Gradient	Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being

Block	Support Notes
	suitable for production code. Thus, blocks suitable for production code remain suitable.
Check Dynamic Gap	Support code generation.
Check Dynamic Lower Bound	
Check Dynamic Range	
Check Dynamic Upper Bound	
Check Input Resolution	Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.
Check Static Gap	
Check Static Lower Bound	
Check Static Range	
Check Static Upper Bound	

Model-Wide Utilities

Block	Support Notes
Block Support Table	Ignored during code generation.
DocBlock	Uses the template symbol you specify for the **Embedded Coder Flag** block parameter to add comments to generated code. Requires an Embedded Coder license. For more information, see "Use a Simulink DocBlock to Add a Comment" on page 36-8.
Model Info	Ignored during code generation.
Timed-Based Linearization	
Trigger-Based Linearization	

Ports & Subsystems

Block	Support Notes
Atomic Subsystem (Simulink)	Support code generation.

Block	Support Notes
CodeReuse Subsystem (Simulink)	
Configurable Subsystem	
Enable	
Enabled Subsystem	
Enabled and Triggered Subsystem	
For Each Subsystem	
For Iterator Subsystem	
Function-Call Feedback Latch	
Function-Call Generator	
Function-Call Split	
Function-Call Subsystem	
If	
If Action Subsystem	
Inport (In1)	
Model	
Model Variants	
Outport (Out1)	
Resettable Subsystem	
Subsystem	
Switch Case	
Switch Case Action Subsystem	
Trigger	
Triggered Subsystem	
Unit System Configuration	
Variant Subsystem	

Block	Support Notes
While Iterator Subsystem	

Signal Attributes

Block	Support Notes
Bus to Vector (Simulink)	Support code generation.
Data Type Conversion	
Data Type Conversion Inherited	
Data Type Duplicate	
Data Type Propagation (Simulink)	
Data Type Scaling Strip	
IC	Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.
Probe	Supports code generation.
Rate Transition	• Supports code generation. • Cannot use inside a triggered subsystem hierarchy.
Signal Conversion	Support code generation.
Signal Specification	
Unit Conversion	
Weighted Sample Time	
Width	

Signal Routing

Block	Support Notes
Bus Assignment	Support code generation.
Bus Creator	
Bus Selector	
Data Store Memory	
Data Store Read	
Data Store Write	
Demux	
Environment Controller	Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.
From	Support code generation.
Goto	
Goto Tag Visibility	
Index Vector	
Manual Switch	Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.
Manual Variant Sink	Support code generation.
Manual Variant Source	

Block	Support Notes
Merge	When multiple signals connected to a Merge block have a non-`Auto` storage class, all non-`Auto` signals connected to that block must *be identically labeled* and *have the same storage class*. When Merge blocks connect directly to one another, these rules apply to all signals connected to Merge blocks in the group.
Multiport Switch	Support code generation.
Mux	
Selector	
State Reader	
State Writer	
Switch	
Variant Sink	
Variant Source	
Vector Concatenate	

Sinks

Block	Support Notes
Display	Ignored for code generation.
Floating Scope (Simulink)	
Outport (Out1)	Supports code generation.
Scope (Simulink)	Ignored for code generation.
Stop Simulation	• Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable. • Generated code stops executing when the stop condition is true.
Terminator	Supports code generation.

Block	Support Notes
To File	Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.
To Workspace	Ignored for code generation.
XY Graph	

Sources

Block	Support Notes
Band-Limited White Noise	Cannot use inside a triggered subsystem hierarchy.
Chirp Signal	Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.
Clock	
Constant	Supports code generation.
Counter Free-Running	Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.

Block	Support Notes
Counter Limited	• The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the **Treat as atomic unit** option. • Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.
Digital Clock	Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.
Enumerated Constant	Supports code generation.
From File	Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.
From Spreadsheet	
From Workspace	Ignored for code generation.
Ground	Support code generation.

Block	Support Notes
Inport (In1)	
Pulse Generator	Cannot use inside a triggered subsystem hierarchy. Does not refer to absolute time when configured for sample-based operation. Depends on absolute time when in time-based operation.
Ramp	Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.
Random Number	Supports code generation.
Repeating Sequence	• Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable. • Consider using the Repeating Sequence Stair or Repeating Sequence Interpolated block instead.
Repeating Sequence Interpolated	• The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the **Treat as atomic unit** option. • Cannot use inside a triggered subsystem hierarchy.
Repeating Sequence Stair	The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In

Block	Support Notes
	certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the **Treat as atomic unit** option.
Signal Builder	Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.
Signal Generator	
Sine Wave	• Depends on absolute time when used inside a triggered subsystem hierarchy. • Does not refer to absolute time when configured for sample-based operation. Depends on absolute time when in time-based operation.
Step	Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.
Uniform Random Number	Supports code generation.
Waveform Generator	Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.

User-Defined

Block	Support Notes
Fcn	Support code generation.
Function Caller	
Initialize Function	
Interpreted MATLAB Function	Consider using the MATLAB Function block instead.
Level-2 MATLAB S-Function	Ignored during code generation.
MATLAB Function	Support code generation.
MATLAB System	
S-Function	S-functions that call into MATLAB are not supported for code generation.
S-Function Builder	
Simulink Function	Support code generation.
Terminate Function	

Simulink Block Data Type Support Table

The Simulink Block Data Type Support table summarizes characteristics of blocks in the Simulink and Fixed-Point Designer block libraries, including whether or not they are recommended for use in production code generation. To view this table, in the MATLAB Command Window, type `showblockdatatypetable`, or consult the block reference pages.

Block Set Support for Code Generation

Several products that include blocks are available for you to consider for code generation. However, before using the blocks for one of these products, consult the documentation for that product to confirm which blocks support code generation.

Modeling Semantic Considerations

In this section...

Data Propagation

The first stage of code generation is compilation of the block diagram. This stage is analogous to that of a C or C++ program. The compiler carries out type checking and preprocessing. Similarly, the Simulink engine verifies that input/output data types of block ports are consistent, line widths between blocks are of expected thickness, and the sample times of connecting blocks are consistent.

The Simulink engine propagates data from one block to the next along signal lines. The data propagated consists of

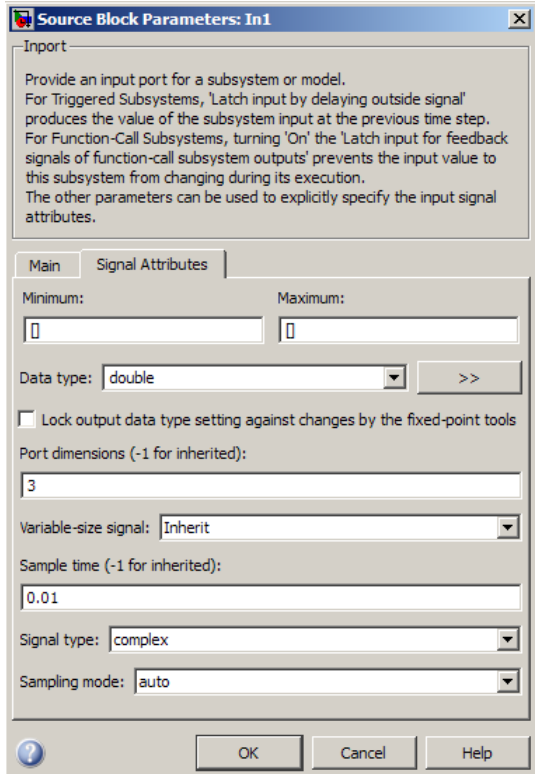
- Data type
- Line widths
- Sample times

You can verify what data types a Simulink block supports by typing

```
showblockdatatypetable
```

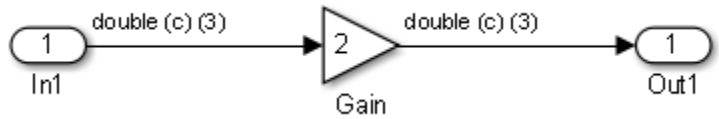
at the MATLAB prompt, or (from the Help browser) clicking the command above.

The Simulink engine typically derives signal attributes from a source block. For example, the Inport block's parameters dialog box specifies the signal attributes for the block.



In this example, the Inport block has a port width of 3, a sample time of .01 seconds, the data type is double, and the signal is complex.

This figure shows the propagation of the signal attributes associated with the Inport block through a simple block diagram.



In this example, the Gain and Outport blocks inherit the attributes specified for the Inport block.

Sample Time Propagation

Inherited sample times in source blocks (for example, a root inport) can sometimes lead to unexpected and unintended sample time assignments. Since a block may specify an inherited sample time, information available at the outset is often insufficient to compile a block diagram completely.

In such cases, the Simulink engine propagates the known or assigned sample times to those blocks that have inherited sample times but that have not yet been assigned a sample time. Thus, the engine continues to fill in the blanks (the unknown sample times) until sample times have been assigned to as many blocks as possible. Blocks that still do not have a sample time are assigned a default sample time.

For a completely deterministic model (one where no sample times are set using the above rules), you should explicitly specify the sample times of your source blocks. Source blocks include root inport blocks and blocks without input ports. You do not have to set subsystem input port sample times. You might want to do so, however, when creating modular systems.

An unconnected input implicitly connects to ground. For ground blocks and ground connections, the sample time is always constant (`inf`).

All blocks have an inherited sample time ($T_s = -1$). They are assigned a sample time of ($T_f - T_i$)/50.

Blocks Whose Outputs Have Constant Values

When you display sample time colors, by default, Constant blocks appear magenta in color to indicate that the block outputs have constant values during simulation. Downstream blocks whose output values are also constant during simulation, such as Gain blocks, similarly appear magenta if they use an inherited sample time. The code generated for these blocks depends in part on the tunability of the block parameters.

If you set **Configuration Parameters > Optimization > Signals and Parameters > Default parameter behavior** to `Inlined`, the block parameters are not tunable in the generated code. Because the block outputs are constant, the code generator eliminates the block code due to constant folding. If the code generator cannot fold the code, or if you select settings to disable constant folding, the block code appears in the model initialization function. The generated code is more efficient because it does not compute the outputs of these blocks during execution.

However, if you configure a block or model so that the block parameters appear in the generated code as tunable variables, the code generator represents the blocks in a different way. Block parameters are tunable if, for example:

- You set **Default parameter behavior** to `Tunable`. By default, numeric block parameters appear as tunable fields of a global parameter structure.
- You use a tunable parameter, such as a `Simulink.Parameter` object that uses a storage class other than `Auto`, as the value of one or more numeric block parameters. These block parameters are tunable regardless of the setting that you choose for **Default parameter behavior**.

If a block parameter is tunable, the generated code must compute the block outputs during execution. Therefore, the block code appears in the model `step` function. If the model uses multiple discrete rates, the block code appears in the output function for the fastest downstream rate that uses the block outputs.

Latches for Subsystem Blocks

When an Inport block is the signal source for a triggered or function-call subsystem, you can use latch options to preserve input values while the subsystem executes. The Inport block latch options include:

For	Use
Triggered subsystems	**Latch input by delaying outside signal**
Function-call subsystems	**Latch input for feedback signals of function-call subsystem outputs**

When you use **Latch input for feedback signals of function-call subsystem outputs** for a function-call subsystem, the code generator

- Preserves latches in generated code regardless of optimizations that might be set
- Places the code for latches at the start of a subsystem's output/update function

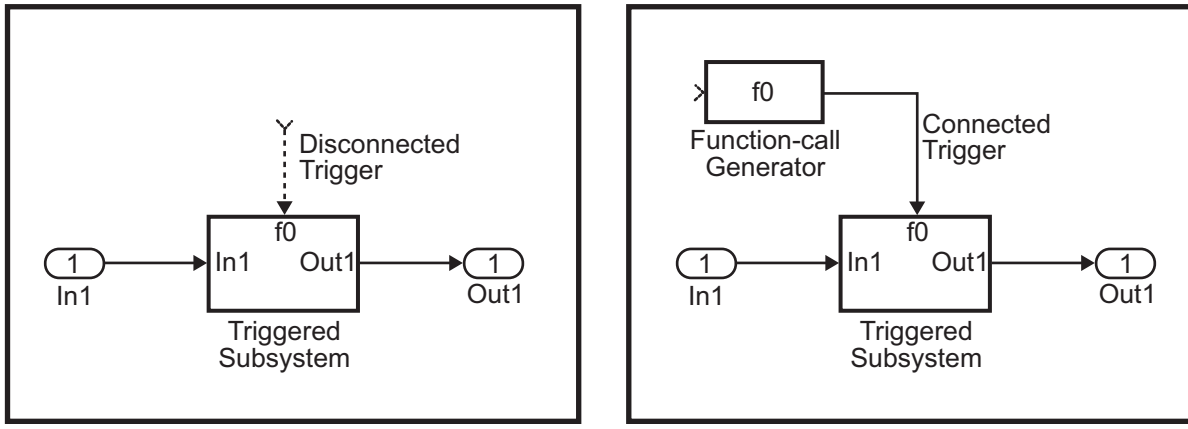
For more information on these options, see the block description of Inport.

Block Execution Order

Once the Simulink engine compiles the block diagram, it creates a *`model`*`.rtw` file (analogous to an object file generated from a C or C++ file). The *`model`*`.rtw` file contains

the connection information of the model, as well as the signal attributes. Thus, the timing engine in can determine when blocks with different rates should be executed.

You cannot override this execution order by directly calling a block (in handwritten code) in a model. For example, in the next figure the `disconnected_trigger` model on the left has its trigger port connected to ground, which can lead to the blocks inheriting a constant sample time. Calling the trigger function, `f()`, directly from user code does not work. Instead, you should use a function-call generator to specify the rate at which `f()` should be executed, as shown in the `connected_trigger` model on the right.



Instead of the function-call generator, you could use another block that can drive the trigger port. Then, you should call the model's main entry point to execute the trigger function.

For multirate models, a common use of the code generator is to generate code for individual models separately and then manually code the I/O between the generated code modules. This approach places the burden of data consistency between models on the developer of the models. Another approach is to let Simulink and the code generator maintain data consistency between rates and generate multirate code for use in a multitasking environment. The Rate Transition block is able to interface periodic and asynchronous signals. For a description of the Simulink Coder block libraries, see "Asynchronous Events" (Simulink Coder). For more information on multirate code generation, see "Modeling for Multitasking Execution" (Simulink Coder).

Algebraic Loops

Algebraic loops are circular dependencies between variables. This prevents the straightforward direct computation of their values. For example, in the case of a system of equations

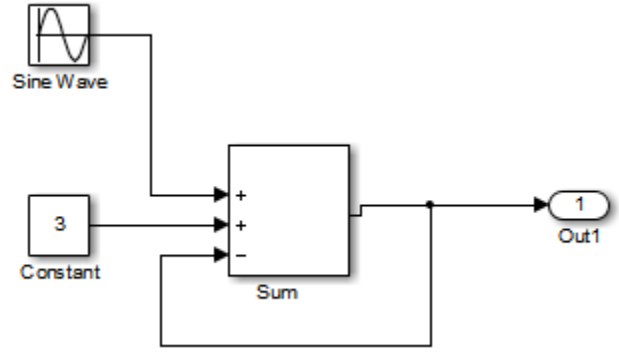
- `x = y + 2`
- `y = -x`

the values of `x` and `y` cannot be directly computed.

To solve this, either repeatedly try potential solutions for `x` and `y` (in an intelligent manner, for example, using gradient based search) or "solve" the system of equations. In the previous example, solving the system into an explicit form leads to

- `2x = 2`
- `y = -x`
- `x = 1`
- `y = -1`

An algebraic loop exists whenever the output of a block having direct feedthrough (such as Gain, Sum, Product, and Transfer Fcn) is fed back as an input to the same block. The Simulink engine is often able to solve models that contain algebraic loops, such as the next diagram.

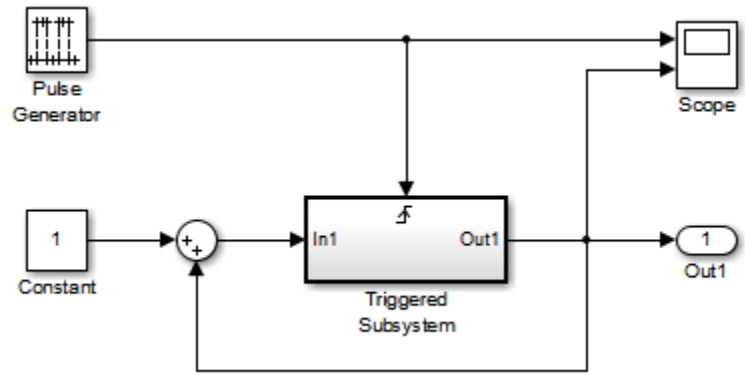


The code generator does not produce code that solves algebraic loops. This restriction includes models that use Algebraic Constraint blocks in feedback paths. However, the

Simulink engine can often eliminate algebraic loops that arise, by grouping equations in certain ways in models that contain them. It does this by separating the update and output functions to avoid circular dependencies. For details, see "Algebraic Loops" (Simulink).

Algebraic Loops in Triggered Subsystems

While the Simulink engine can minimize algebraic loops involving atomic and enabled subsystems, a special consideration applies to some triggered subsystems. An example for which code can be generated is shown in the following model and triggered subsystem.



The default Simulink behavior is to combine output and update methods for the subsystem, which creates an apparent algebraic loop, even though the Unit Delay block in the subsystem has no direct feedthrough.

You can allow the Simulink engine to solve the problem by splitting the output and update methods of triggered and enabled-triggered subsystems when feasible. If you want the code generator to take advantage of this feature, select the **Minimize algebraic loop occurrences** check box in the Subsystem Parameters dialog box. Select this option to avoid algebraic loop warnings in triggered subsystems involved in loops.

Note: If you check this box, the generated code for the subsystem might contain split output and update methods, even if the subsystem is not actually involved in a loop. Also, if a direct feedthrough block (such as a Gain block) is connected to the inport in the

above triggered subsystem, the Simulink engine cannot solve the problem, and the code generator is unable to generate code.

A similar **Minimize algebraic loop occurrences** option appears on the **Model Referencing** pane of the Configuration Parameters dialog box. Selecting it enables the Simulink Coder software to generate code for models containing Model blocks that are involved in algebraic loops.

Modeling Guidelines for Blocks

Code generation modeling guidelines include recommended model settings, block usage, and block parameters. When you develop models for code generation, use these guidelines.

For more information, see "Modeling Guidelines" (Simulink).

Code Generation Modeling Guidelines	"cgsl_0101: Zero-based indexing" (Simulink) "cgsl_0102: Evenly spaced breakpoints in lookup tables" (Simulink) "cgsl_0103: Precalculated signals and parameters" (Simulink) "cgsl_0104: Modeling global shared memory using data stores" (Simulink) "cgsl_0105: Modeling local shared memory using data stores" (Simulink) "cgsl_0201: Redundant Unit Delay and Memory blocks" (Simulink)

See Also

"Modeling Guidelines for Subsystems" on page 2-36 | "Modeling Guidelines for Charts" on page 2-38 | "Modeling Guidelines for MATLAB Functions" on page 2-40 | "Modeling Guidelines for Model Configuration" on page 2-41

Modeling Guidelines for Subsystems

When you develop models and generate code for subsystems, use the modeling guideline recommendations.

For more information, see "Modeling Guidelines" (Simulink).

Code Generation Modeling Guidelines	"cgsl_0204: Vector and bus signals crossing into atomic subsystems or Model blocks" (Simulink)
High-Integrity Systems Modeling Guidelines	"hisl_0009: Usage of For Iterator Subsystem blocks" (Simulink) "hisl_0010: Usage of If blocks and If Action Subsystem blocks" (Simulink) "hisl_0011: Usage of Switch Case blocks and Action Subsystem blocks" (Simulink) "hisl_0023: Verification of model and subsystem variants" (Simulink)
MathWorks Automotive Advisory Board (MAAB) Control Algorithm Guidelines	db_0040: Model hierarchy (Simulink) db_0042: Port block in Simulink models (Simulink) db_0081: Unconnected signals, block inputs and block outputs (Simulink) db_0143: Similar block types on the model levels (Simulink) db_0144: Use of Subsystems (Simulink) db_0146: Triggered, enabled, conditional Subsystems (Simulink) jc_0111: Direction of Subsystem (Simulink) jc_0201: Usable characters for Subsystem names (Simulink) jc_0231: Usable characters for block names (Simulink) jc_0281: Naming of Trigger Port block and Enable Port block (Simulink)

	jc_0321: Trigger layer (Simulink) jc_0331: Structure layer (Simulink) jc_0351: Methods of initialization (Simulink) jm_0002: Block resizing (Simulink) na_0005: Port block name visibility in Simulink models (Simulink) na_0006: Guidelines for mixed use of Simulink and Stateflow (Simulink) na_0008: Display of labels on signals (Simulink) na_0009: Entry versus propagation of signal labels (Simulink) na_0012: Use of Switch vs. If-Then-Else Action Subsystem (Simulink)

See Also

"Modeling Guidelines for Blocks" on page 2-35 | "Modeling Guidelines for Charts" on page 2-38 | "Modeling Guidelines for MATLAB Functions" on page 2-40 | "Modeling Guidelines for Model Configuration" on page 2-41

Modeling Guidelines for Charts

When you develop models and generate code for charts, use the modeling guideline recommendations.

For more information, see “Modeling Guidelines” (Simulink).

High-Integrity Systems Modeling Guidelines	“hisf_0001: Mealy and Moore semantics” (Simulink) “hisf_0002: User-specified state/transition execution order” (Simulink) “hisf_0009: Strong data typing (Simulink and Stateflow boundary)” (Simulink) “hisf_0011: Stateflow debugging settings” (Simulink) “hisf_0003: Usage of bitwise operations” (Simulink) “hisf_0004: Usage of recursive behavior” (Simulink) “hisf_0007: Usage of junction conditions (maintaining mutual exclusion)” (Simulink) “hisf_0010: Usage of transition paths (looping out of parent of source and destination objects)” (Simulink) “hisf_0012: Chart comments” (Simulink) “hisf_0013: Usage of transition paths (crossing parallel state boundaries)” (Simulink) “hisf_0014: Usage of transition paths (passing through states)” (Simulink) “hisf_0015: Strong data typing (casting variables and parameters in expressions)” (Simulink)
MathWorks Automotive Advisory Board (MAAB) Control	db_0127: MATLAB commands in Stateflow (Simulink) db_0151: State machine patterns for transition actions (Simulink)

Algorithm Guidelines	jc_0451: Use of unary minus on unsigned integers in Stateflow (Simulink) jc_0481: Use of hard equality comparisons for floating point numbers in Stateflow (Simulink) jc_0501: Format of entries in a State block (Simulink) jc_0511: Setting the return value from a graphical function (Simulink) jc_0521: Use of the return value from graphical functions (Simulink) jc_0531: Placement of the default transition (Simulink) jc_0541: Use of tunable parameters in Stateflow (Simulink) jm_0011: Pointers in Stateflow (Simulink) jm_0012: Event broadcasts (Simulink) na_0001: Bitwise Stateflow operators (Simulink) na_0013: Comparison operation in Stateflow (Simulink)

See Also

"Modeling Guidelines for Blocks" on page 2-35 | "Modeling Guidelines for Subsystems" on page 2-36 | "Modeling Guidelines for MATLAB Functions" on page 2-40 | "Modeling Guidelines for Model Configuration" on page 2-41

Modeling Guidelines for MATLAB Functions

When you develop models and generate code for MATLAB Functions, use the modeling guideline recommendations.

For more information, see "Modeling Guidelines" (Simulink).

High-Integrity Systems Modeling Guidelines	"himl_0001: Usage of standardized MATLAB function headers" (Simulink) "himl_0002: Strong data typing at MATLAB function boundaries" (Simulink) "himl_0003: Limitation of MATLAB function complexity" (Simulink) "himl_0005: Usage of global variables in MATLAB functions" (Simulink)

See Also

"Modeling Guidelines for Blocks" on page 2-35 | "Modeling Guidelines for Subsystems" on page 2-36 | "Modeling Guidelines for Charts" on page 2-38 | "Modeling Guidelines for Model Configuration" on page 2-41

Modeling Guidelines for Model Configuration

When you develop models and generate code, use the modeling guideline configuration recommendations.

For more information, see “Modeling Guidelines” (Simulink).

Code Generation Modeling Guidelines	“cgsl_0301: Prioritization of code generation objectives for code efficiency” (Simulink) “cgsl_0302: Diagnostic settings for multirate and multitasking models” (Simulink)
High-Integrity Systems Modeling Guidelines	“hisl_0043: Configuration Parameters > Diagnostics > Solver” (Simulink) “hisl_0044: Configuration Parameters > Diagnostics > Sample Time” (Simulink) “hisl_0301: Configuration Parameters > Diagnostics > Compatibility” (Simulink) “hisl_0302: Configuration Parameters > Diagnostics > Data Validity > Parameters” (Simulink) “hisl_0303: Configuration Parameters > Diagnostics > Merge block” (Simulink) “hisl_0304: Configuration Parameters > Diagnostics > Model initialization” (Simulink) “hisl_0305: Configuration Parameters > Diagnostics > Debugging” (Simulink) “hisl_0306: Configuration Parameters > Diagnostics > Connectivity > Signals” (Simulink) “hisl_0307: Configuration Parameters > Diagnostics > Connectivity > Buses” (Simulink) “hisl_0308: Configuration Parameters > Diagnostics > Connectivity > Function calls” (Simulink)

	"hisl_0309: Configuration Parameters > Diagnostics > Type Conversion" (Simulink) "hisl_0310: Configuration Parameters > Diagnostics > Model Referencing" (Simulink) "hisl_0311: Configuration Parameters > Diagnostics > Stateflow" (Simulink)

See Also

"Modeling Guidelines for Blocks" on page 2-35 | "Modeling Guidelines for Subsystems" on page 2-36 | "Modeling Guidelines for Charts" on page 2-38 | "Modeling Guidelines for MATLAB Functions" on page 2-40

Subsystems in Simulink Coder

Code Generation of Subsystems

For you to control how code is generated for a nonvirtual subsystem, the code generator provides subsystem parameters that you can use. The categories of nonvirtual subsystems are:

- *Conditionally executed* subsystems. Execution depends upon a control signal or control block. These subsystems include:
  - Triggered
  - Enabled
  - Action
  - Iterator
  - Function-call

  For more information, see "Conditional Subsystems" (Simulink).
- *Atomic* subsystems: A virtual subsystem can be declared atomic (and therefore nonvirtual) by using the "Treat as atomic unit" (Simulink) parameter in the Subsystem Parameters dialog box.

For more information on nonvirtual subsystems and atomic subsystems, see "Systems and Subsystems" (Simulink) and open the Subsystem Semantics library.

You can design and configure your model to control the code generated from nonvirtual subsystems.

To...	See...
Generate inlined code from a selected nonvirtual subsystem.	"Inline Subsystem Code" on page 3-7
Generate code for only a subsystem.	"Generate Code and Executables for Individual Subsystem" on page 3-4
Generate separate functions with no arguments, and optionally place the subsystem code in a separate file.	"Generate Subsystem Code as Separate Function and Files" on page 3-10
Generate a single reentrant function for a subsystem that is included in multiple places within a model.	"Generate Reusable Function for Identical Subsystems Within a Model" on page 3-11

To...	See...
Generate a single reentrant function for a subsystem that is included in multiple places in a model reference hierarchy.	"Generate Reusable Code for Subsystems Shared Across Models" on page 3-28 and "Code Reuse For Subsystems Shared Across Models" on page 3-20

Note: If you generate code for a virtual subsystem, code generator treats the subsystem as atomic and generates the code accordingly. The resulting code can change the execution behavior of your model, for example, by applying algebraic loops, and therefore introduce inconsistencies with the simulation behavior. Declare virtual subsystems as atomic subsystems, which makes simulation and execution behavior consistent for your model consistent.

Subsystem Code Dependence

Code generated from nonvirtual subsystems may or may not be completely independent of the generating model. When generating code for a subsystem, the code may reference global data structures of the model, even if the subsystem code is in a separate file. Each subsystem code file contains `include` directives and comments describing the dependencies. The code generator checks for cyclic file dependencies and warns about them at build time. For descriptions of how generated code is packaged, see "Manage Build Process File Dependencies" (Simulink Coder).

To generate subsystem code that is independent of the generating model, place the subsystem in a library and configure it as a reusable subsystem. For more information, see "Code Reuse For Subsystems Shared Across Models" on page 3-20.

Generate Code and Executables for Individual Subsystem

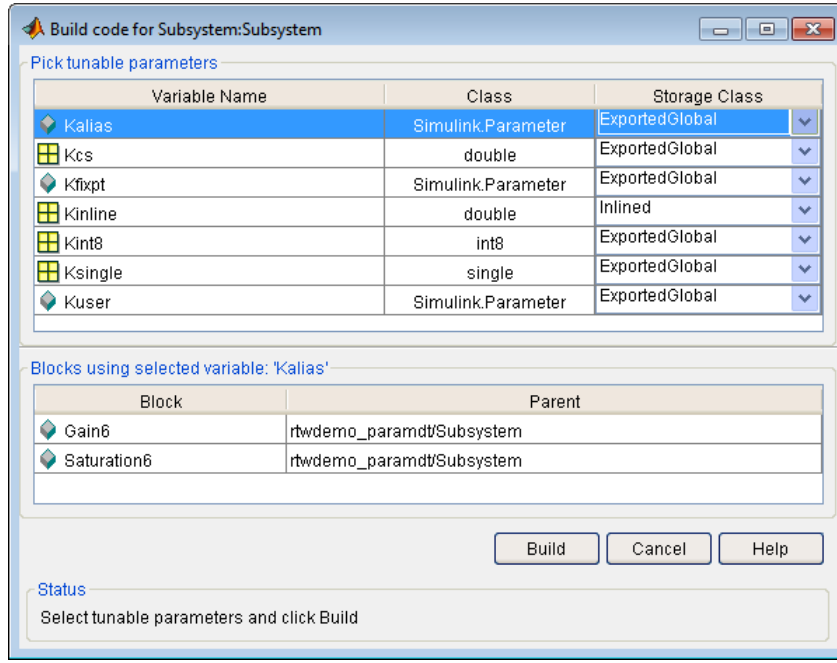
You can generate code and build an executable for a subsystem within a model. The code generation and build process uses the code generation and build parameters of the root model.

1. In the Configuration Parameters dialog box, set up the code generation and build parameters, similar to setting up the code generation for a model.
2. Right-click the Subsystem block. From the context menu, select **C/C++ Code > Build This Subsystem** from the context menu.

   Alternatively, in the current model, click a subsystem and then from the **Code** menu, select **C/C++ Code > Build Selected Subsystem**.

   Note When you select **Build This Subsystem**, if the model is operating in external mode, the build process automatically turns off external mode for the duration of the build. The code generator restores external mode upon completion of the build process.
3. The **Build code for Subsystem** window displays a list of the subsystem parameters. The upper pane displays the name, class, and storage class of each variable (or data object) that is referenced as a block parameter in the subsystem. When you select a parameter in the upper pane, the lower pane shows the blocks that reference the parameter and the parent system of each block.

   The **Storage Class** column contains a menu for each row. The menu options set the storage class or inline the parameter. To declare a parameter to be tunable, set the **Storage Class** to a value other than **Inlined**.



For more information on tunable and inlined parameters and storage classes, see "Block Parameter Representation in the Generated Code" (Simulink Coder).

4 After selecting tunable parameters, **Build** to initiate the code generation and build process.

5 The build process displays status messages in the MATLAB Command Window. When the build is complete, the generated executable is in your working folder. The name of the generated executable is *`subsystem`*`.exe` (on PC platforms) or *`subsystem`* (on The Open Group UNIX® platforms). *`subsystem`* is the name of the source subsystem block.

The generated code is in a build subfolder, named *`subsystem_target`*`_rtw`. *`subsystem`* is the name of the source subsystem block and *`target`* is the name of the target configuration.

When you generate code for a subsystem, you can generate an S-function by selecting **Code > C/C++ Code> Generate S-Function**, or you right-click the subsystem block and select **C/C++ Code > Build This Subsystem** from the context menu. For more

information on S-functions, see "Automate S-Function Generation with S-Function Builder" (Simulink Coder).

Subsystem Build Limitations

The following limitations apply to building subsystems:

- Subsystem build does not support a subsystem that has a function-call trigger input or a function-call output.
- When you right-click a subsystem block and select **C/C++ Code > Build This Subsystem** from the context menu to build a subsystem that includes an Outport block for which the **Data type** parameter specifies a bus object, you must address errors that result from setting signal labels. To configure the software to display these errors, in the Configuration Parameters dialog box for the parent model, on the **Diagnostics > Connectivity** pane, set the **Signal label mismatch** parameter to `error`.
- When a subsystem is in a triggered or function-call subsystem, the right-click build process might fail if the subsystem code is not sample-time independent. To find out whether a subsystem is sample-time independent:

  1. Copy all blocks in the subsystem to an empty model.
  2. In the Configuration Parameters dialog box, on the **Solver** pane, set:
     a. **Type** to `Fixed-step`.
     b. **Periodic sample time constraint** to `Ensure sample time independent`.
     c. Click **Apply**.
  3. Update the model. If the model is sample-time dependent, Simulink generates an error in the process of updating the diagram.

Inline Subsystem Code

You can configure a nonvirtual subsystem to inline the subsystem code with the model code. In the Subsystem Parameters dialog box, setting the **Function packaging** parameter to `Auto` or `Inline` inlines the generated code of the subsystem.

The `Auto` option is the default. When there is only one instance of a subsystem in the model, the `Auto` option inlines the subsystem code. When multiple instances of a subsystem exist, the `Auto` option results in a single copy of the function (as a reusable function). For function-call subsystems with multiple callers, the subsystem code is generated as if you specified `Nonreusable function`.

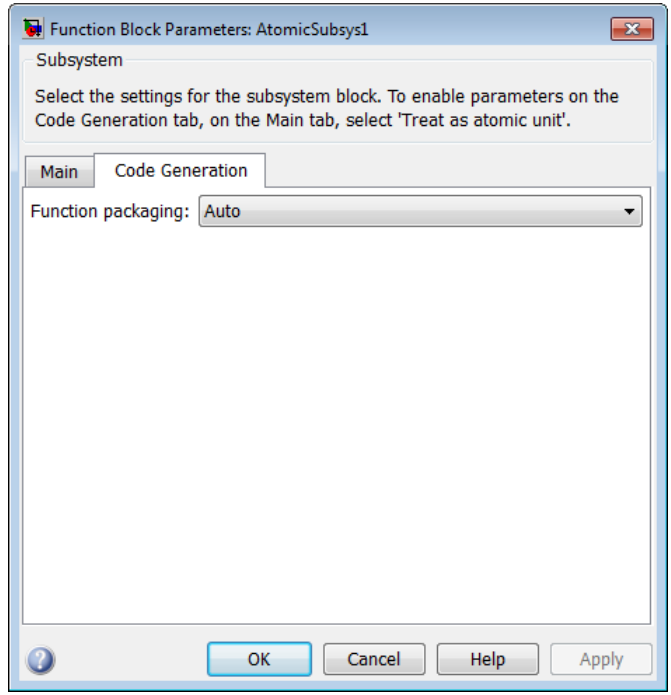
To inline subsystem code, select `Inline`. The `Inline` option explicitly directs the code generator to inline the subsystem unconditionally.

Configure Subsystem to Inline Code

To configure your subsystem for inlining:

1. Right-click the Subsystem block. From the context menu, select **Block Parameters (Subsystem)**.
2. In the Subsystem Parameters dialog box, if the subsystem is virtual, select **Treat as atomic unit**. This option makes the subsystem nonvirtual. On the **Code Generation** tab, the **Function packaging** option is now available.

   If the system is already nonvirtual, the **Function packaging** option is already selected.
3. Click the **Code Generation** tab and select `Auto` or `Inline` from the **Function packaging** parameter.



4 Click **Apply** and close the dialog box.

The border of the subsystem thickens, indicating that it is nonvirtual.

When you generate code from your model, the code generator inlines subsystem code within *model*`.c` or *model*`.cpp` (or in its parent system's source file). You can identify this code by system/block identification tags, such as:

```
/* Atomic SubSystem Block: <Root>/AtomicSubsys1 */
```

Exceptions to Inlining

There are certain cases in which the code generator does not inline a nonvirtual subsystem, even though the **Inline** option is selected.

- If the subsystem is a function-call subsystem that is called by a noninlined S-function, the **Inline** option is ignored. Noninlined S-functions make calls by using function

pointers. Therefore, the function-call subsystem must generate a function with all arguments present.

- In a feedback loop involving function-call subsystems, the code generator forces one of the subsystems to be generated as a function instead of inlining it. Based on the order in which the subsystems are sorted internally, the software selects the subsystem to be generated as a function.
- If a subsystem is called from an S-function block that sets the option `SS_OPTION_FORCE_NONINLINED_FCNCALL` to `TRUE`, it is not inlined. When user-defined Async Interrupt blocks or Task Sync blocks are present, this result might occur. Such blocks must be generated as functions. These blocks are located in the `vxlib1` block library and use the `SS_OPTION_FORCE_NONINLINED_FCNCALL` option. This library demonstrates integration with an example RTOS (VxWorks®).[1]

Note: You can use the blocks in the `vxlib1` (Simulink Coder) library (Async Interrupt and Task Sync) for simulation and code generation. These blocks provide starting point examples to help you develop custom blocks for your target environment.

1. VxWorks is a registered trademark of Wind River® Systems, Inc.

Generate Subsystem Code as Separate Function and Files

To generate both a separate subsystem function and a separate file for a subsystem in a model:

1. Right-click a Subsystem block. From the context menu, select **Block Parameters (Subsystem)**.
2. In the Subsystem Parameters dialog box, if the subsystem is virtual, select **Treat as atomic unit**. On the **Code Generation** tab, the **Function packaging** parameter is now available.
3. Click the **Code Generation** tab and select `Nonreusable function` from the **Function packaging** parameter. The `Nonreusable function` option enables two parameters:
   - The "Function name options" (Simulink) parameter controls the naming of the generated function.
   - The "File name options" (Simulink) parameter controls the naming of the generated file.
4. Set the **Function name options** parameter.
5. Set the **File name options** parameter to a value other than `Auto`. If you are generating a reusable function for your subsystem, see "Generate Reusable Function for Identical Subsystems Within a Model" on page 3-11 or "Generate Reusable Code for Subsystems Shared Across Models" on page 3-28.
6. Click **Apply** and close the dialog box.

Generate Reusable Function for Identical Subsystems Within a Model

In the Subsystem Parameters dialog box, the **Function packaging** parameter option `Nonreusable function` generates functions that use global data. The `Reusable function` option generates reusable functions that have data passed as arguments (enabling them to be reentrant). Selecting `Reusable function` generates a function with arguments that allows the subsystem code to be shared by other instances of it in the model. This action supports less code instead of replicating the code for each instance of a subsystem or each time it is called.

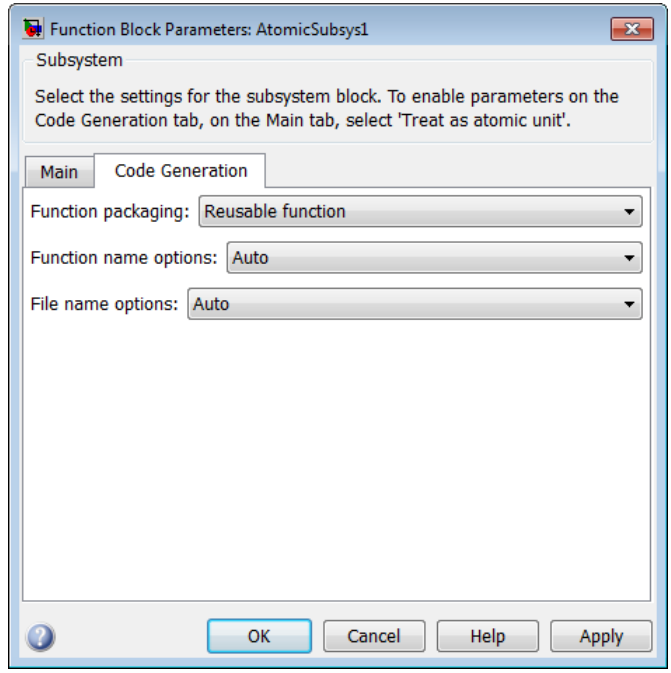
To determine reusability of the subsystem code, the code generator performs a checksum to determine if subsystems are identical. The generated function has arguments, for example, for block inputs and outputs (`rtB_*`), continuous states (`rtDW_*`), parameters (`rtP_*`).

Note: In the generated code, the call interface is subject to change from release to release. Therefore, do not directly call reusable functions from external code.

To generate one reusable function for identical subsystems within a model:

1. Right-click the Subsystem block. From the context menu, select **Block Parameters (Subsystem)**.
2. In the Subsystem Parameters dialog box, if the subsystem is virtual, select **Treat as atomic unit**. On the **Code Generation** tab, the **Function packaging** menu is now available.

   If the subsystem is already nonvirtual, the **Function packaging** menu is already selected.
3. Click the **Code Generation** tab and select `Reusable function` for the **Function packaging** parameter.



For more information about this setting, see "Considerations for Function Packaging Options Auto and Reusable function" on page 3-13.

4 Set the function name using the "Function name options" (Simulink) parameter.

Note: If you do not choose `Auto`, for other Subsystem blocks that you want to share this code, specify the same function name for those Subsystem blocks.

---

5 Set the file name using the "File name options" (Simulink) parameter to a value other than `Auto`. If your generated code is under source control, a value other than `Auto` prevents the generated file name from changing due to unrelated model modifications.

Note: For other Subsystem blocks that you want to share this code, specify the same file name for those Subsystem blocks.

---

6 Click **Apply** and close the dialog box.

For a summary of code reuse limitations, see "Code Reuse Limitations for Subsystems" on page 3-17.

Considerations for Function Packaging Options `Auto` and `Reusable function`

When you want multiple instances of a subsystem to be represented as one reusable function, you can designate each one of them as `Auto` or as `Reusable function`. Use one or the other, because using both creates two reusable functions, one for each specification. The outcomes of these choices differ only when reuse is not possible. Selecting `Auto` does not allow control of the function or file name for the subsystem code.

The `Reusable function` and `Auto` options both try to determine if multiple instances of a subsystem exist and if the code can be reused. When reuse is not possible, there are differences in the options behavior:

- `Auto` yields inlined code. If circumstances prohibit inlining, then the generated code is separate functions for each subsystem instance.
- `Reusable function` yields a separate function with arguments for each subsystem instance in the model.

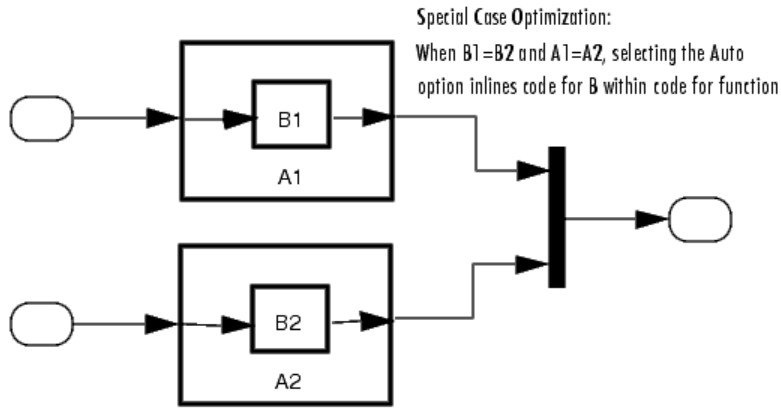
Code Reuse for Subsystems with Mask Parameters

The code generator can produce reusable (reentrant) code for a model containing identical atomic subsystems. Selecting the `Reusable function` option for **Function packaging** enables such code reuse, and causes a single function with arguments to be generated that is called when an identical atomic subsystem executes. See "Subsystems" (Simulink Coder) for details and restrictions on the use of this option.

Mask parameters become arguments to reusable functions. However, for reuse to occur, each instance of a reusable subsystem must declare the same set of mask parameters. If, for example subsystem A has mask parameters `b` and `K`, and subsystem B has mask parameters `c` and `K`, then code reuse is not possible, and the code generator produces separate functions for A and B.

Optimize Code for Identical Nested Subsystems

The **Function packaging** parameter `Auto` option can optimize code in situations in which identical subsystems contain other identical subsystems, by both reusing and inlining generated code. Suppose a model, such as the one shown in Reuse of Identical Nested Subsystems, contains identical subsystems `A1` and `A2`. `A1` contains subsystem `B1`, and `A2` contains subsystem `B2`, which are identical. In such cases, the `Auto` option causes one function to be generated which is called for both `A1` and `A2`. This function contains one piece of inlined code to execute `B1` and `B2`. This optimization generates less code which improves execution speed.



Reuse of Identical Nested Subsystems

Generate Reusable Code for Subsystems Containing S-Function Blocks

There are several requirements that need to be met in order for subsystems containing S-function blocks to be reused. For the list of requirements, see "S-Functions That Support Code Reuse" (Simulink Coder).

When you select the `Reusable function` option, two additional options are enabled, **Function name options** and **File name options**. If you use these fields to enter a function name and/or a file name, you must specify exactly the same function name and file name for each instance of identical subsystems for the code generator to reuse the subsystem code. For an example, follow the procedure in "Generate Reusable Function for Identical Subsystems Within a Model" on page 3-11.

Generate Reusable Code from Stateflow Charts

You can generate reusable code from a Stateflow chart, or from a subsystem containing a chart, *except* when the Stateflow chart contains exported graphical functions.

Code Reuse Limitations for Subsystems

The code generator uses a checksum to determine whether subsystems are identical and reusable. Subsystem code is not reused, if:

- In blocks and data objects, you use symbols to specify dimensions.
- A port used by multiple instances of a subsystem has different sample times, data types, complexity, frame status, or dimensions across the instances.
- The output of a subsystem is marked as a global signal.
- Subsystems contain identical blocks with different names or parameter settings.
- The output of a subsystem is connected to a Merge block, and the output of the Merge block is a custom storage class that is implemented in the C code as memory that is nonaddressable (for example, `BitField`).
- The input of a subsystem is nonscalar and has a custom storage class that is implemented in the C code as memory that is nonaddressable.
- A masked subsystem has a parameter that is nonscalar and has a custom storage class that is implemented in the C code as memory that is nonaddressable.
- A function-call subsystem uses mask parameters of any kind when you set the model configuration parameter "Default parameter behavior" (Simulink) to `Tunable`. To reuse the masked function-call subsystem, you can place the masked subsystem inside a new atomic subsystem without a mask, and move the Trigger block from the masked subsystem into the atomic subsystem.
- A block in the subsystems uses a *partially tunable expression*. Some partially tunable expressions can disable code reuse.

  Partially tunable expressions are expressions that contain one or more tunable variables in addition to an expression that is not tunable. For example, suppose that you create the tunable variable `K` with value `15.23` and the tunable variable `P` with value `[5;7;9]`. The expression `K+P'` is a partially tunable expression because the expression `P'` is not tunable. For more information about tunable expression limitations, see "Tunable Expression Limitations" (Simulink Coder).

If you select `Reusable function`, and the code generator determines that you cannot reuse the code for a subsystem, it generates a separate function that is not reused. The code generation report might show that the separate function is reusable, even if only one subsystem uses it. If you prefer that subsystem code be inlined in such circumstances rather than deployed as functions, choose `Auto` for the **Function packaging** option.

Blocks That Prevent Code Reuse

Use of the following blocks in a subsystem can also prevent the subsystem code from being reused:

- Scope blocks (with data logging enabled)
- S-Function blocks that fail to meet certain criteria (see "S-Functions That Support Code Reuse" (Simulink Coder))
- To File blocks (with data logging enabled)
- To Workspace blocks (with data logging enabled)

Code Reuse Limitations for Subsystems Shared Across Referenced Models

The code generator uses a checksum to determine whether reusable library subsystems are identical. The code generator places the reusable library subsystem code in the shared utilities folder, and the reusable code is independent of the generated code of the top model or the referenced model. For example, the reusable library subsystem code does not include *`model`*`.h` or *`model`*`_types.h`.

Reusable code that is generated to the shared utilities folder and is dependent on the model code does not compile. If the code generator determines that the reusable library subsystem code is dependent on the model code, the reusable subsystem code is not generated to the shared utilities folder. The following cases can generate code that is dependent on the model code, when the reusable library subsystem:

- Contains a block that uses time-related functionality, such as a Step block, or continuous time or multirate blocks.
- Contains one or more Model blocks.
- Contains subsystems that are not inlined or a reusable library subsystem.
- Contains a signal that is not an `Auto` storage class. Variables of non-`Auto` storage classes are generated to *`model`*`.h`.
- Contains a parameter that is not an `Auto` storage class.
- Contains a user-defined type where **Data Scope** is not set to `Exported`. The code generator might place the type definition in *`model`*`_types.h`.
- Is a variant subsystem that generates preprocessor conditionals. Preprocessor directives defining the variant objects are included in *`model`*`_types.h`.

Related Examples

- "Determine Why Subsystem Code Is Not Reused" on page 3-36

Code Reuse For Subsystems Shared Across Models

To reuse common functionality, you can include multiple instances of a subsystem:

- Within a single model, which is a top model or part of model reference hierarchy
- Across multiple referenced models in a model reference hierarchy
- Across multiple top models that contain Model blocks
- Across multiple top models that do not include Model blocks

To generate a reusable function for a subsystem which is included in multiple models:

- If the subsystem is in a model reference hierarchy, set the configuration parameter, "Shared code placement" (Simulink Coder) to `Auto`. Otherwise, for each model that uses the subsystem, set **Shared code placement** to `Shared location`. The **Shared code placement** parameter is in the Configuration Parameters dialog box, on the **Code Generation > Interface** pane.
- The subsystem must be defined in a library and configured for reuse. This subsystem is referred to as a *reusable library subsystem*. For more information, see "Reusable Library Subsystem" on page 3-21.

For an example, see "Generate Reusable Code for Subsystems Shared Across Models" on page 3-28.

The code generator performs a checksum to determine reusability. There are cases when the code generator cannot reuse subsystem code. For more information, see "Code Reuse Limitations for Subsystems" on page 3-17.

Related Examples

- "Determine Why Subsystem Code Is Not Reused" on page 3-36

Reusable Library Subsystem

A reusable library subsystem is a subsystem included in a library that is configured for reuse. The Subsystem parameters must be set as follows:

- **Treat as an atomic unit** is selected.
- On the **Code Generation** tab:
  - **Function packaging** is set to `Reusable function`.
  - **Function name options**

    and **File name options** are set to `Auto` or `Use subsystem name`.

For more information on creating a library, see "Libraries" (Simulink). For an example of creating a reusable library subsystem, see "Generate Reusable Code for Subsystems Shared Across Models" on page 3-28.

Code Generation of a Reusable Library Subsystem

For incremental code generation, if the reusable library subsystem changes, a rebuild of itself and its parents occurs. During the build, if a matching function is not found, a new instance of the reusable function is generated into the shared utilities folder. If a different matching function is found from previous builds, that function is used, and a new reusable function is not emitted.

For subsequent builds, unused files are not replaced or deleted from your folder. During development of a model, when many obsolete shared functions exist in the shared utilities folder, you can delete the folder and regenerate the code. If all instances of a reusable library subsystem are removed from a model reference hierarchy and you regenerate the code, the obsolete shared functions remain in the shared utilities folder until you delete them.

If a model changes such that the change might cause different generated code for the subsystem, a new reusable function is generated. For example, model configuration parameters that modify code comments might cause different generated code for the subsystem even if the reusable library subsystem did not change.

Reusable Library Subsystem Code Placement and Naming

The code generator uses checksums to determine reusability. The generated code of a reusable library subsystem is independent of the generated code of the model. Code for the reusable library subsystem is generated to the shared utility folder, `slprj/target/_sharedutils`, instead of the model reference hierarchy folders. The generated code for the supporting types, which are generated to the `.h` file, are also in the shared utilities folder.

For unique naming, reusable function names have a checksum appended to the reusable library subsystem name. For example, the code and files for a subsystem, `SS1`, which links to a reusable library subsystem, `RLS`, might be:

- Function name: `RLS_mgdjlngd`
- File name: `RLS_mgdjlnd.c` and `RLS_mgdjlnd.h`

Reusable Library Subsystem in the Top Model

In a model reference hierarchy, if an instance of the reusable library subsystem is in the top model, then on the **Model Referencing** pane of the Configuration Parameters dialog box, you must select the **Pass fixed-size scalar root input by value for code generation** parameter. If you do not select the parameter, a separate shared function is generated for the reusable library subsystem instance in the top model, and a reusable function is generated for instances in the referenced models.

Reusable Library Subsystem Connected to Root Outport

If a reusable library subsystem is connected to the root outport, reuse does not happen with identical subsystems that are not connected to the root outport. However, you can set **Pass reusable system outputs as** to `Individual arguments` on the **Optimizations > Signals and Parameters** pane to make sure that reuse occurs between these subsystems. This parameter requires an Embedded Coder license.

Code Generation of Constant Parameters

The code generator attempts to generate constant parameters to the shared utilities folder first. If constant parameters are not generated to the shared utilities folder, they are defined in the top model in a global constant parameter structure. The declaration of the structure, `ConstParam_`*`model`*, is in *`model`*`.h`:

```
/* Constant parameters (auto storage) */
typedef struct {
   /* Expression: [1 2 3 4 5 6 7]
    * Referenced by: '<Root>/Constant'
    */
   real_T Constant_Value[7];

   /* Expression: [7 6 5 4 3 2 1]
    * Referenced by: '<Root>/Gain'
    */
   real_T Gain_Gain[7];
 } ConstParam_model;
```

The definition of the constant parameters, *`model`*`_constP`, is in:

```
/* Constant parameters (auto storage) */
const ConstParam_model model_ConstP = {
   /* Expression: [1 2 3 4 5 6 7]
    * Referenced by: '<Root>/Constant'
    */
   { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0 },

   /* Expression: [7 6 5 4 3 2 1]
    * Referenced by: '<Root>/Gain'
    */
   { 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0 }
};
```

The *`model`*`_constP` is passed as an argument to referenced models. For more information on how shared constants are generated, see “Shared Constant Parameters for Code Reuse” on page 3-24.

Shared Constant Parameters for Code Reuse

You can share the generated code for constant parameters across models if:

- Constant parameters are shared in a model reference hierarchy, or
- On the **Code Generation > Interface** pane, the model configuration parameter "Shared code placement" (Simulink Coder) is set to `Shared location`.

If you do not want to generate shared constants, and **Shared code placement** is set to `Shared location`, set the parameter `GenerateSharedConstants` to `off`. For example, to turn off shared constants for the current model, in the Command Window, type the following.

```
set_param(gcs,'GenerateSharedConstants','off');
```

The shared constant parameters are generated individually to the `const_params.c` file, which is placed in the shared utilities folder `slprj/`*`target`*`/_sharedutils`.

For example, if a constant has multiple uses within a model reference hierarchy where the top model is named `topmod`, the code for the shared constant is as follows:

- In the shared utility folder, `slprj/grt/_sharedutils`, the constant parameters are defined in `const_params.c` and named `rtCP_pooled_` appended to a unique checksum:

  ```
  extern const real_T rtCP_pooled_lfcjjmohiecj[7];
  const real_T rtCP_pooled_lfcjjmohiecj[7] = { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0 };

  extern const real_T rtCP_pooled_ppphohdbfcba[7];
  const real_T rtCP_pooled_ppphohdbfcba[7] = { 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0 };
  ```

- In *`top_model`*`_private.h` or in a referenced model, *`ref_model`*`_private.h`, for better readability, the constants are renamed as follows:

  ```
  extern const real_T rtCP_pooled_lfcjjmohiecj[7];
  extern const real_T rtCP_pooled_ppphohdbfcba[7];

  #define rtCP_Constant_Value   rtCP_pooled_lfcjjmohiecj  /* Expression: [1 2 3 4 5 6 7]
                                                           * Referenced by: '<Root>/Constant'*/
  #define rtCP_Gain_Gain        rtCP_pooled_ppphohdbfcba  /* Expression: [7 6 5 4 3 2 1]
                                                           * Referenced by: '<Root>/Gain' */
  ```

- In `topmod.c` or `refmod.c`, the call site might be:

  ```
  for (i = 0; i < 7; i++) {
     topmod_Y.Out1[i] = (topmod_U.In1 + rtCP_Constant_Value[i]) * rtCP_Gain_Gain[i];
  }
  ```

The code generator attempts to generate all constants as individual constants to the `const_params.c` file in the shared utilities folder. Otherwise, constants are generated as described in "Code Generation of Constant Parameters" on page 3-23.

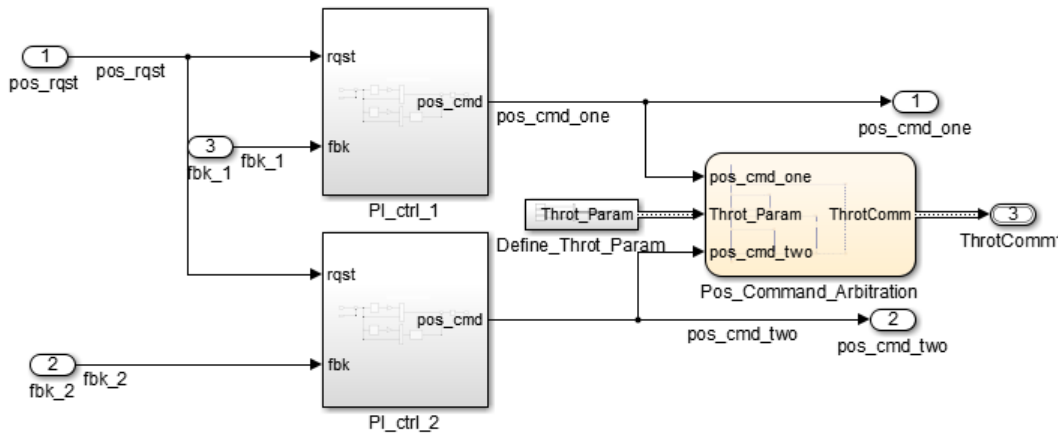
Suppress Shared Constants in the Generated Code

You can choose whether or not the code generator produces shared constants and shared functions. You may want to be able to keep the code and data separate between subsystems, or you may find that sharing constants results in a memory shortage during code generation.

You can change this parameter programmatically using the parameter `GenerateSharedConstants` with `set_param` and `get_param`.

In the following example, when `GenerateSharedConstants` is set to `on`, the code generator defines the constant values in the `_sharedutils` folder in the `const_params.c` file. When `GenerateSharedConstants` is set to `off`, the code generator defines the constant values in a nonshared area, in the `model_ert_rtw` file in the `model_data.c` file.

Open the model `rtwdemo_throttlecntrl`:



In the Configuration parameters dialog box, on the **Code Generation** > **Interface** pane, verify that "Shared code placement" (Simulink Coder) is set to `Shared location`. If

Shared code placement is set to `Auto`, the `GenerateSharedConstants` setting is ignored. If you try to set the parameter value, an error message appears. The default value of `GenerateSharedConstants` is `on`.

In the Command Window, set `GenerateSharedConstants` to `on`:

```
>> set_param('rtwdemo_throttlecntrl','GenerateSharedConstants','on')
```

You see the shared constant definitions in the folder `slprj/grt/_sharedutils`, in the file `const_params.c`:

```
extern const real_T rtCP_pooled_H4eTKtECwveN[9];
const real_T rtCP_pooled_H4eTKtECwveN[9] = { 1.0, 0.75, 0.6, 0.0, 0.0, 0.0, 0.6,
  0.75, 1.0 } ;

extern const real_T rtCP_pooled_SghuHxKVKGHD[9];
const real_T rtCP_pooled_SghuHxKVKGHD[9] = { -1.0, -0.5, -0.25, -0.05, 0.0, 0.05,
  0.25, 0.5, 1.0 } ;

extern const real_T rtCP_pooled_WqWb2t17NA2R[7];
const real_T rtCP_pooled_WqWb2t17NA2R[7] = { -1.0, -0.25, -0.01, 0.0, 0.01, 0.25,
  1.0 } ;

extern const real_T rtCP_pooled_YgnalOwM3c14[7];
const real_T rtCP_pooled_YgnalOwM3c14[7] = { 1.0, 0.25, 0.0, 0.0, 0.0, 0.25, 1.0
} ;
```

In the Command Window, set `GenerateSharedConstants` to `off`:

```
>> set_param('rtwdemo_throttlecntrl','GenerateSharedConstants','off')
```

You can see the unshared constants in the folder `rtwdemo_throttlecntrl_grt_rtw`, in the file `rtwdemo_throttlecntrl_data.c`:

```
/* Constant parameters (auto storage) */
const ConstP_rtwdemo_throttlecntrl_T rtwdemo_throttlecntrl_ConstP = {
  /* Pooled Parameter (Expression: P_OutMap)
   * Referenced by:
   *   '<S2>/Proportional  Gain Shape'
   *   '<S3>/Proportional  Gain Shape'
   */
  { 1.0, 0.25, 0.0, 0.0, 0.0, 0.25, 1.0 },

  /* Pooled Parameter (Expression: P_InErrMap)
   * Referenced by:
   *   '<S2>/Proportional  Gain Shape'
   *   '<S3>/Proportional  Gain Shape'
   */
  { -1.0, -0.25, -0.01, 0.0, 0.01, 0.25, 1.0 },

  /* Pooled Parameter (Expression: I_OutMap)
   * Referenced by:
```

```
   *   '<S2>/Integral  Gain Shape'
   *   '<S3>/Integral  Gain Shape'
   */
  { 1.0, 0.75, 0.6, 0.0, 0.0, 0.0, 0.6, 0.75, 1.0 },

  /* Pooled Parameter (Expression: I_InErrMap)
   * Referenced by:
   *   '<S2>/Integral  Gain Shape'
   *   '<S3>/Integral  Gain Shape'
   */
  { -1.0, -0.5, -0.25, -0.05, 0.0, 0.05, 0.25, 0.5, 1.0 }
};
```

Shared Constant Parameters Limitations

No shared constants or shared functions are generated for a model when:

- The model has a Code Replacement Library (CRL) that is specified for data alignment.
- The model is specified to replace data type names in the generated code.
- The **Memory Section** for constants is `MemVolatile` or `MemConstVolatile`.
- The parameter `GenerateSharedConstants` is set to `off`.

Individual constants are not shared, if:

- A constant is referenced by a non-inlined S-function.
- A constant has a user-defined type where **Data Scope** is not set to `Exported`.

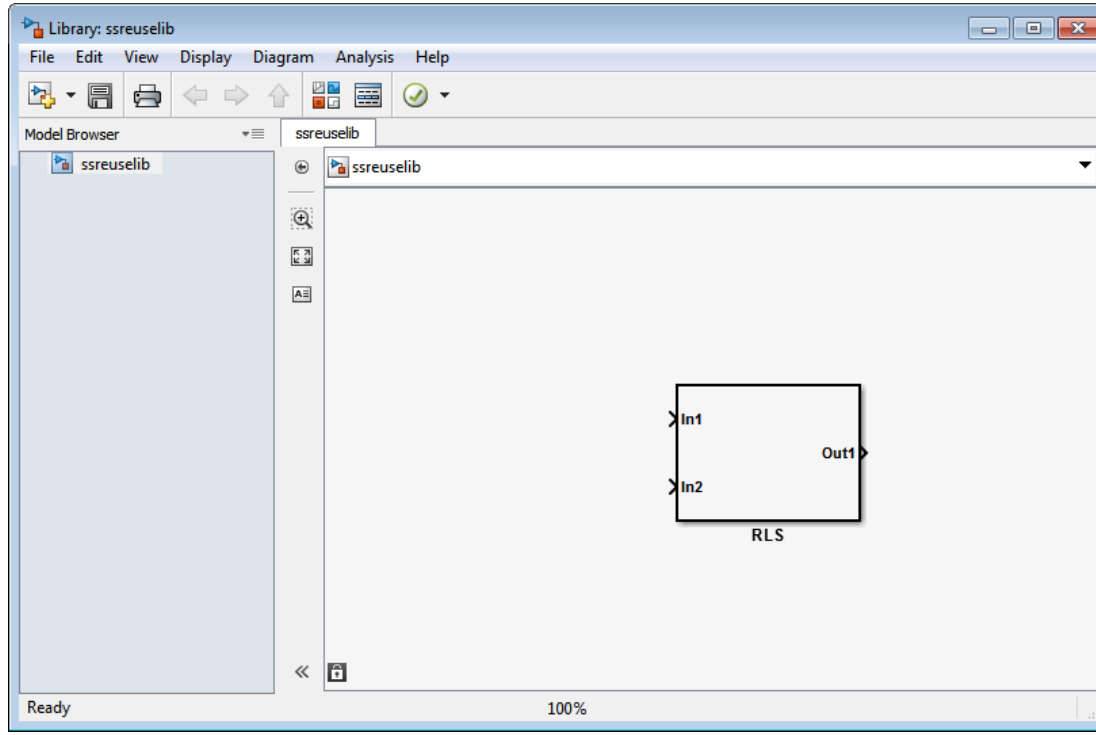
Generate Reusable Code for Subsystems Shared Across Models

This example shows how to configure a reusable library subsystem and generate a reusable function for a subsystem shared across referenced models. The result is reusable code for the subsystem, which is generated to the shared utility folder (`slprj/target/_sharedutils`).

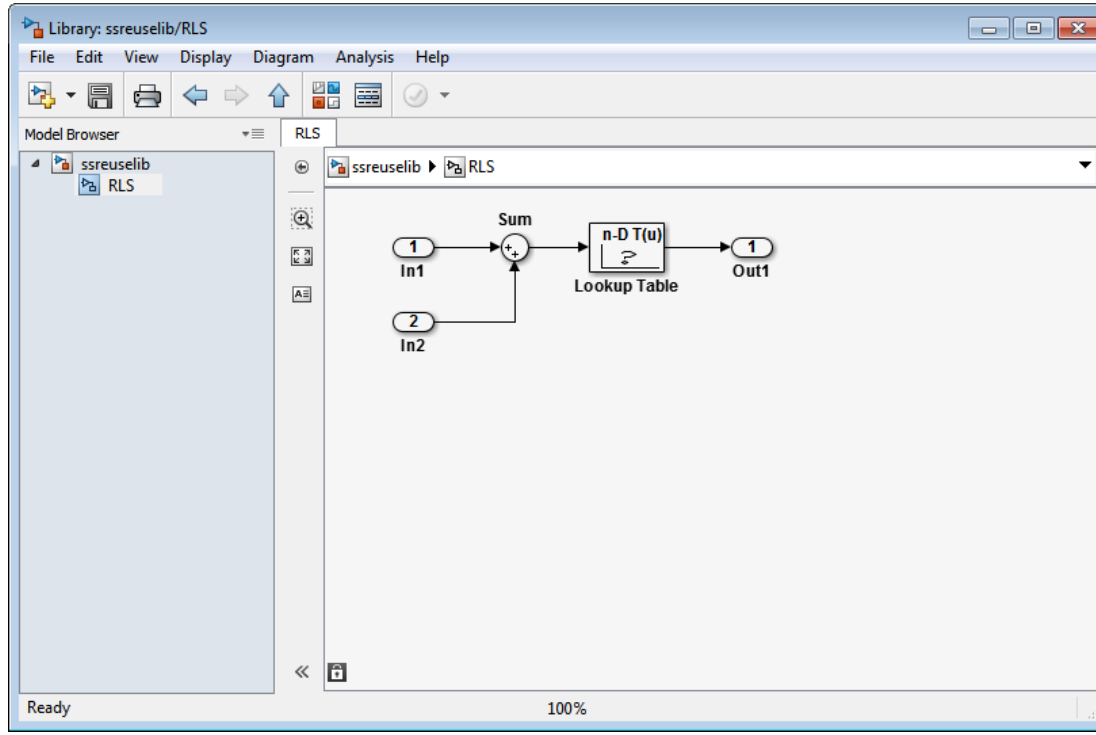
In this section...

Create a reusable library subsystem.

1 In the Simulink Editor, select **File > New > Library**. Open `rtwdemo_ssreuse` to copy and paste subsystem `SS1` into the Library Editor. This action loads the variables for `SS1` into the base workspace. Rename the subsystem block to `RLS`.



2 Click the Subsystem block and press **Ctrl+U** to view the contents of subsystem RLS.



3 To configure the subsystem, in the Library editor, right-click `RLS`. In the context menu, select **Block Parameters(Subsystem)**. In the Subsystem Parameters dialog box, choose the following options:

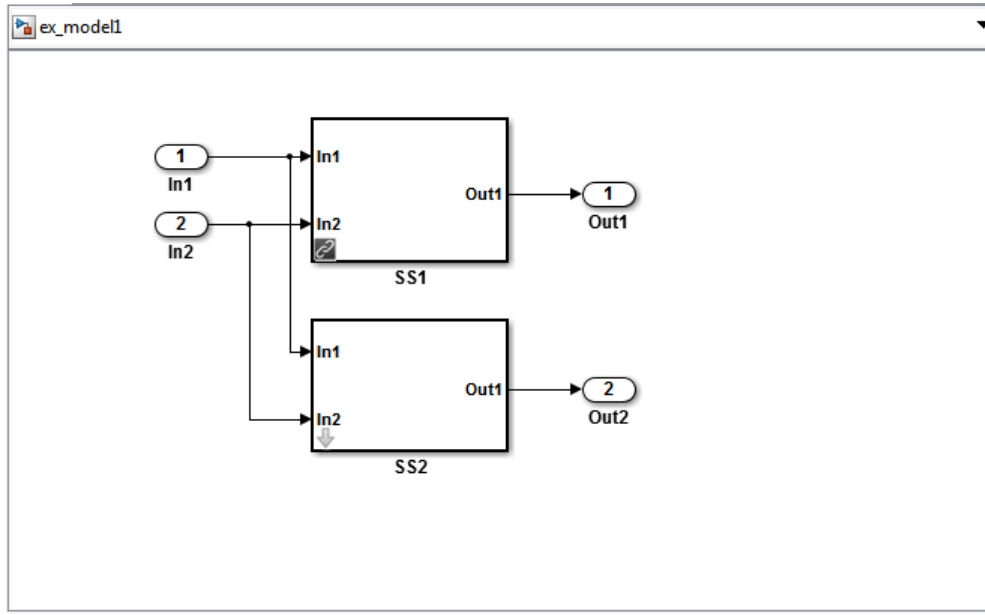
   - Select **Treat as an atomic unit**.
   - On the **Code Generation** tab:
     - Set **Function packaging** to `Reusable function`.
     - Set **Function name options** and **File name options** to `Auto`.

4 Click **Apply** and **OK**.

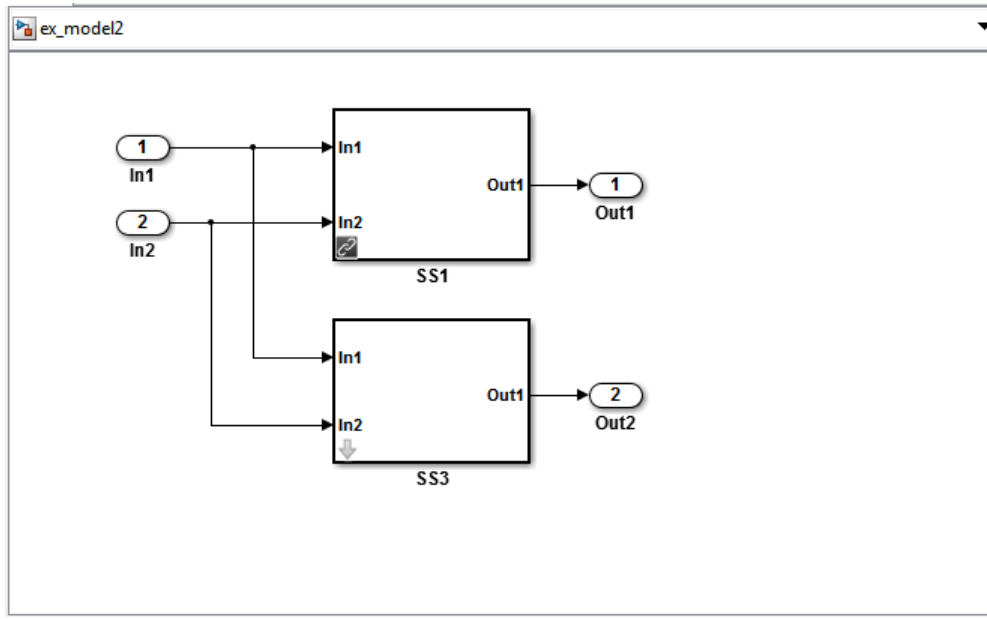
5 Save the reusable library subsystem as `ssreuselib`, which creates a file, `ssreuselib.slx`.

Create the example model.

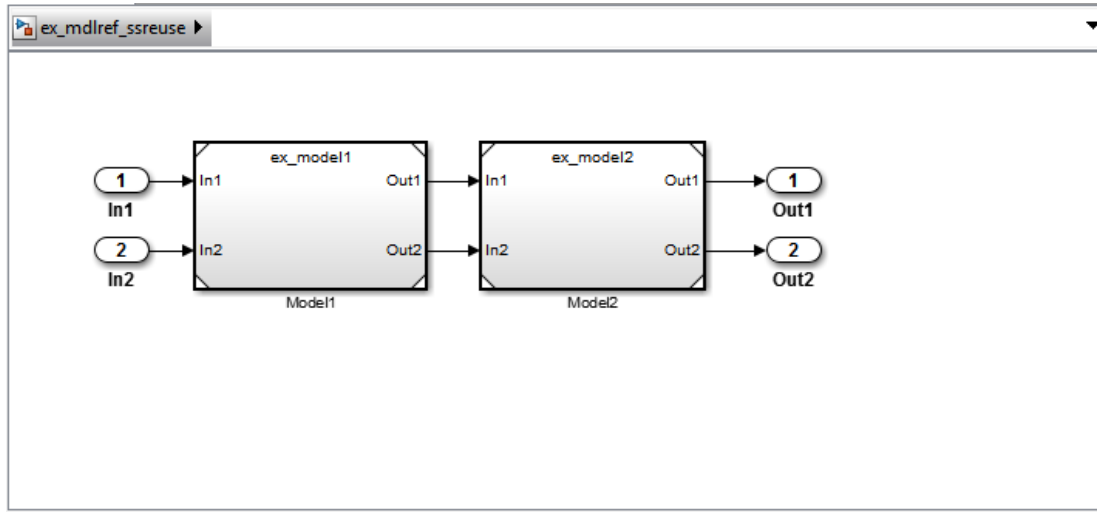
1 Create a model which includes one instance of `RLS` from `ssreuselib`. Name this subsystem `SS1`. Add another subsystem and name it `SS2`. Name the model `ex_model1`.



2 Create another model which includes one instance of `RLS` from `ssreuselib`. Name this subsystem `SS1`. Add another subsystem and name it `SS3`. Name the model `ex_model2`.



3 Create a top model with two model blocks that reference `ex_model1` and `ex_model2`. Save the top model as `ex_mdlref_ssreuse`.



Set configuration parameters of the top model.

1. With model `ex_mdlref_ssreuse` open in the Simulink Editor, select **Simulation > Model Configuration Parameters** to open the Configuration Parameters dialog box.
2. On the **Solver** pane, specify the **Type** as `Fixed-step`.
3. On the **Model Referencing** pane, select **Pass fixed-size scalar root inputs by value for code generation**.
4. On the **Code Generation > Report** pane, select **Create code generation report** and **Open report automatically**.
5. On the **Code Generation > Interface** pane, set the "Shared code placement" (Simulink Coder) to `Shared location`.
6. On the **Code Generation > Symbols** pane, set the **Maximum identifier length** to `256`. This step is optional.
7. Click **Apply** and **OK**.

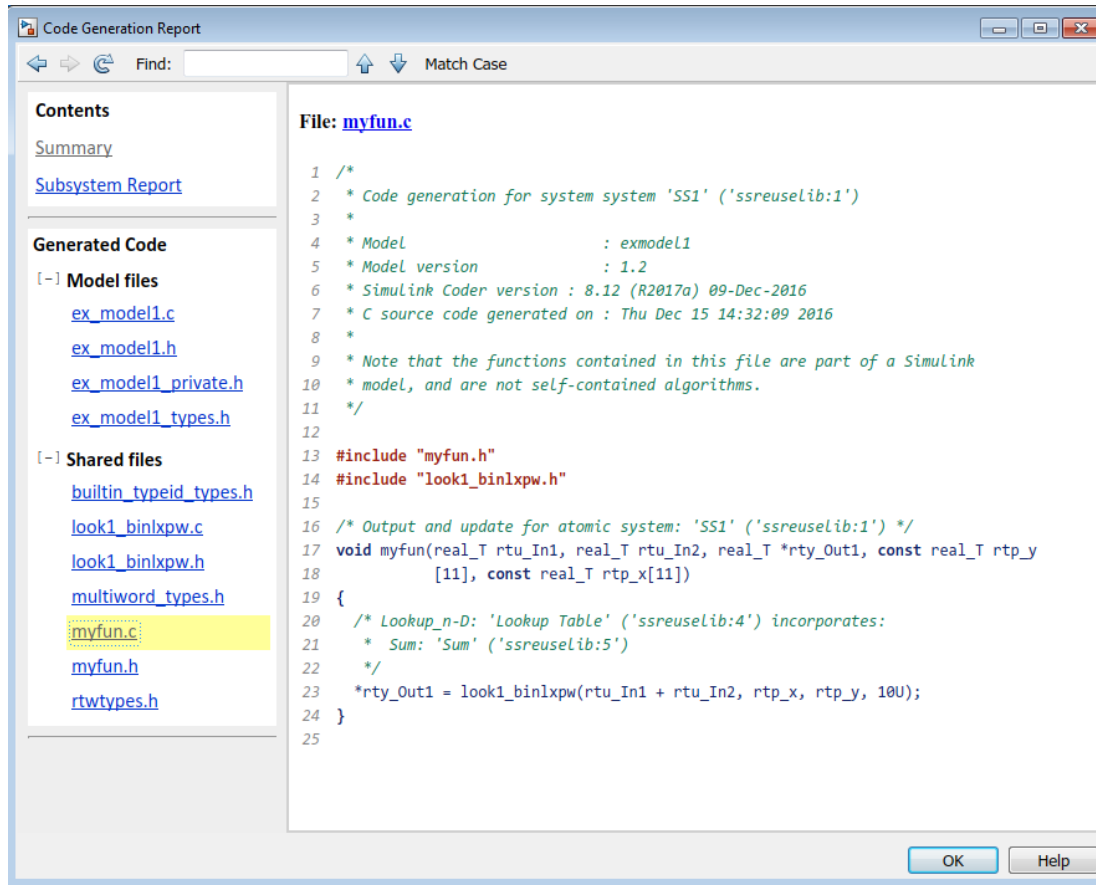
Create and propagate a configuration reference.

1. In the Simulink Editor, select **View > Model Explorer** to open the Model Explorer. In the left navigation column of the Model Explorer, expand the `ex_mdlref_ssreuse` node.
2. Right-click **Configuration** and select `Convert to Configuration Reference`.
3. In the Convert Active Configuration to Reference dialog box, click **OK**. This action converts the model configuration set to a configuration reference, `Simulink.ConfigSetRef`, and creates the configuration reference object, `configSetObj`, in the base workspace.
4. In the left navigation column, right-click **Reference (Active)** and select `Propagate to Referenced Models`.
5. In the Configuration Reference Propagation to Referenced Models dialog box, select the referenced models in the list. Click **Propagate**.

Now, the top model and referenced models use the same configuration reference, `Reference (Active)`, which points to a model configuration reference object, `configSetObj`, in the base workspace. When you save your model, you also need to export the `configSetObj` to a MAT-file.

Generate and view the code.

1. To generate code, in the Simulink Editor, press **Ctrl-B**. After the code is generated, the code generation report opens.
2. To view the code generation report for a referenced model, in the left navigation pane, in the **Referenced Models** section, select `ex_model1`. The code generation report displays the generated files for `ex_model1`.
3. In the left navigation pane, expand the **Shared files**. The code generator uses the reusable library subsystem name. The code for subsystem `SS1` is in `myfun.c` and `myfun.h`.



4 Click **Back** and navigate to the `ex_model2` code generation report. `ex_model2` uses the same source code, `myfun.c` and `myfun.h`, as the code for `ex_model1`. Your subsystem function and file names will be different.

Related Examples

- "Determine Why Subsystem Code Is Not Reused" on page 3-36
- "Generate Reusable Function for Identical Subsystems Within a Model" on page 3-11
- "Enable Component Reuse with Clone Detection" (Simulink Verification and Validation)

More About

- "Code Generation of Subsystems" on page 3-2
- "Code Reuse For Subsystems Shared Across Models" on page 3-20
- "Code Reuse Limitations for Subsystems" on page 3-17
- "Libraries" (Simulink)

Determine Why Subsystem Code Is Not Reused

Due to the limitations described in "Code Reuse Limitations for Subsystems" on page 3-17, the code generator might not reuse generated code as you expect. To determine why code generated for a subsystem is not reused, see "Review Subsystems Section of HTML Code Generation Report" on page 3-36. If you cannot determine why based on the report, see "Compare Subsystem Checksum Data" on page 3-36.

Review Subsystems Section of HTML Code Generation Report

If you determine that the code generator does not generate code for a subsystem as reusable code, and you specified the subsystem as reusable, examine the Subsystems section of the HTML code generation report (see "Generate a Code Generation Report" (Simulink Coder)). The Subsystems section contains:

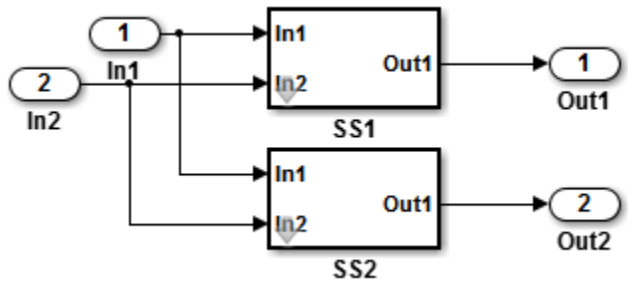
- A table that summarizes how nonvirtual subsystems were converted to generated code.
- Diagnostic information that describes why the contents of some subsystems were not generated as reusable code.

The Subsystems section also indicates the mapping of each noninlined subsystem in the model to functions or reused functions in the generated code. For an example, open and build the `rtwdemo_atomic` model.

Compare Subsystem Checksum Data

You can determine why subsystem code is not reused by accessing and comparing subsystem checksum data. The code generator determines whether subsystems are identical by comparing subsystem checksums, as noted in "Code Reuse Limitations for Subsystems" on page 3-17. For subsystem reuse across referenced models, this procedure might not catch every difference.

Consider the model, `rtwdemo_ssreuse`. `SS1` and `SS2` are instances of the same subsystem. In both instances the subsystem parameter **Function packaging** is set to `Reusable function`.



Use the method, `Simulink.SubSystem.getChecksum`, to get the checksum for a subsystem. Compare the results to determine why code is not reused.

1. Open the model `rtwdemo_ssreuse`. Save a copy of the model in a folder where you have write access.
2. In the model window, select subsystem `SS1`. In the command window, enter

   ```
   SS1 = gcb;
   ```
3. In the model window, select subsystem `SS2`. In the command window, enter

   ```
   SS2 = gcb;
   ```
4. Use the method, `Simulink.SubSystem.getChecksum`, to get the checksum for each subsystem. This method returns two output values: the checksum value and details on the input used to compute the checksum.

   ```
   [chksum1, chksum1_details] = ...
   Simulink.SubSystem.getChecksum(SS1);
   [chksum2, chksum2_details] = ...
   Simulink.SubSystem.getChecksum(SS2);
   ```
5. Compare the two checksum values. They should be equal based on the subsystem configurations.

   ```
   isequal(chksum1, chksum2)
   ans =
        1
   ```
6. To see how you can use `Simulink.SubSystem.getChecksum` to determine why the checksums of two subsystems differ, change the data type mode of the output port of SS1 so that it differs from that of SS2.

a Look under the mask of SS1. Right-click the subsystem. In the context menu, select **Mask > Look Under Mask**.

b In the block diagram of the subsystem, double-click the Lookup Table block to open the Subsystem Parameters dialog box.

c Click **Data Types**.

d Select **Saturate on integer overflow** and click **OK**.

7 Get the checksum for SS1. Compare the checksums for the two subsystems. This time, the checksums are not equal.

```
[chksum1, chksum1_details] = ...
Simulink.SubSystem.getChecksum(SS1);
isequal(chksum1, chksum2)
ans =
     0
```

8 After you determine that the checksums are different, find out why. The Simulink engine uses information, such as signal data types, some block parameter values, and block connectivity information, to compute the checksums. To determine why checksums are different, you compare the data used to compute the checksum values. You can get this information from the second value returned by `Simulink.SubSystem.getChecksum`, which is a structure array with four fields.

Look at the structure `chksum1_details`.

```
chksum1_details

chksum1_details =
          ContentsChecksum: [1x1 struct]
         InterfaceChecksum: [1x1 struct]
     ContentsChecksumItems: [287x1 struct]
    InterfaceChecksumItems: [53x1 struct]
```

`ContentsChecksum` and `InterfaceChecksum` are component checksums of the subsystem checksum. The remaining two fields, `ContentsChecksumItems` and `InterfaceChecksumItems`, contain the checksum details.

9 Determine whether a difference exists in the subsystem contents, interface, or both. For example:

```
isequal(chksum1_details.ContentsChecksum.Value,...
        chksum2_details.ContentsChecksum.Value)
ans =
```

```
     0
isequal(chksum1_details.InterfaceChecksum.Value,...
        chksum2_details.InterfaceChecksum.Value)
ans =
     1
```

In this case, differences exist in the contents.

10 Write a script like the following to find the differences.

```
idxForCDiffs=[];
for idx = 1:length(chksum1_details.ContentsChecksumItems)
    if (~strcmp(chksum1_details.ContentsChecksumItems(idx).Identifier, ...
                chksum2_details.ContentsChecksumItems(idx).Identifier))
            disp(['Identifiers different for contents item ', num2str(idx)]);
            idxForCDiffs=[idxForCDiffs, idx];
    end
    if (ischar(chksum1_details.ContentsChecksumItems(idx).Value))
            if (~strcmp(chksum1_details.ContentsChecksumItems(idx).Value, ...
                        chksum2_details.ContentsChecksumItems(idx).Value))
            disp(['Character vector values different for contents item ', num2str(idx)]);
            idxForCDiffs=[idxForCDiffs, idx];
            end
    end
    if (isnumeric(chksum1_details.ContentsChecksumItems(idx).Value))
            if (chksum1_details.ContentsChecksumItems(idx).Value ~= ...
                        chksum2_details.ContentsChecksumItems(idx).Value)
            disp(['Numeric values different for contents item ', num2str(idx)]);
            idxForCDiffs=[idxForCDiffs, idx];
            end
    end
end
```

11 Run the script. The following example assumes that you named the script `check_details`.

```
check_details
Character vector values different for contents item 202
```

The results indicate that differences exist for index item 202 in the subsystem contents.

12 Use the returned index values to get the handle, identifier, and value details for each difference found.

```
chksum1_details.ContentsChecksumItems(202)

ans =

        Handle: 'rtwdemo_ssreuse/SS1/Lookup Table'
    Identifier: 'SaturateOnIntegerOverflow'
         Value: 'on'
```

The details identify the Lookup Table block parameter **Saturate on integer overflow** as the focus for debugging a subsystem reuse issue.

4

Code Generation of Functions and Function Callers in Simulink Coder

Modeling Functions and Callers for Code Generation

In this section...

Functions and Callers

Use a Simulink Function block and a Function Caller block to instruct the code generator to generate C functions and function calls in the generated code for encapsulation and portability. A Simulink Function block is a nonreusable subsystem.

With a Simulink Function block and a Function Caller block, you can:

- Use nested function calls to call a function from a function.
- Choose to separate function definitions and calls into different models.
- Specify SIL and PIL simulations.
- Integrate code for multiple top models where the Simulink Function block and Function callers are in different models.
- Use global data to communicate between a server and its parent model. This data uses custom storage classes to customize how the data is communicated.
- Model a client and server application using the export functions modeling style.

Simulink Function blocks and Function Caller blocks do not honor the `MaxStackSize` parameter.

For more information, see "Simulink Functions" (Simulink), "Diagnostics Using a Client-Server Architecture" (Simulink), and Simulink Function block.

Input and Output Arguments

When you set up your model that contains Simulink Function blocks for code generation:

- Do not define the signals entering and leaving Argument Inport blocks and Argument Outport blocks in the Simulink Function definition with a storage class.

- Do not specify Argument Inport and Argument Outport blocks as test points.
- If you specify the data type of signals entering and leaving Argument Inport and Argument Outport blocks as a `Simulink.IntEnumType`, `Simulink.AliasType` or `Simulink.Bus` type, then you must specify the arguments as `Imported` or `Exported`, not `Auto`.
- The Simulink Function block and the Function Caller blocks must agree in data type, complexity, dimension, and number of arguments.

For more information, see Argument Inport and Argument Outport.

Function and Function Caller Definitions Across Models

You can define a Simulink Function block and Function Caller block in different models. When the code generator generates code for a model hierarchy, it can encounter either a Simulink Function block or a Function Caller block first. If the code generator finds the Simulink Function block first, the software uses the function definition from the Simulink Function block. If the code generator then encounters a Function Caller block that does not match the function definition, the code generator issues an error. This error prompts you to either change the Function Caller block to match the Simulink Function block or remove the `slprj` folder. Verify that the arguments and data types in the Function Caller blocks match the arguments and data types in the Simulink Function block. Regenerate code for the models involved.

If the code generator encounters a Function Caller block first, then the code generator uses the function definition derived from the Function Caller block. If the code generator then encounters a Simulink Function block with different arguments and data types from the Function Caller block, the code generator issues a warning message. Verify that the Function Caller blocks match the Simulink Function block. Regenerate the code where you have made changes.

Specifying two Simulink Function blocks with the same name is an error. Modify one of the blocks and remove the `slprj` folder.

Code Generation Files

In the build folders, the code generator creates different files depending on the setting you choose.

When one of the following is true, the code generator creates files as shown in the table.

- The system target file is `grt.tlc`.
- The system target file is `ert.tlc` and you do not have an Embedded Coder license.
- The system target file is `ert.tlc` and you have an Embedded Coder license. For the **Code Generation** > **Code Placement** > **File packaging format** parameter, select `Modular`.

For a model named `model` and a function named `fn1`, the code generator creates files with modular file packaging.

Modular File Packaging of Files for GRT System Target or ERT System Target

File	Folder	Contents
`model.c`	`model_`*`target`*`_rtw`	Calls to the function.
`model.h`	`model_`*`target`*`_rtw`	This header file includes declarations and header files for the function, including `fn1.h` and `fn1_private.h`.
`fn1.c`	`model_`*`target`*`_rtw`	Code for the function.
`fn1.h`	`slprj/`*`target`*`/_sharedutils`	This header file contains the `fn1` function prototype declaration. This header file is included in the code generated for the Function Caller blocks associated with the function. The `fn1.h` file placement is not affected by the **Code Generation** > **Interface** > **Shared code placement** parameter value.
`fn1_private.h`	`model_`*`target`*`_rtw`	This header file includes declarations and header files for the function, including `fn1.h`.

If you have an Embedded Coder license, set the system target file to `ert.tlc`, and set the **File packaging format** parameter to `Compact` or `Compact (with separate data file)`, the code generator creates files with compact file packaging.

Compact File Packaging of Files for ERT System Target

File	Folder	Contents
`model.c`	`model_ert_rtw`	Calls to the function and code for the function.

File	Folder	Contents
`model.h`	`model_ert_rtw`	This header file includes declarations and header files for the function, including `fn1.h`.
`fn1.h`	`slprj\ert\_sharedutils`	This header file contains the `fn1` function prototype declaration. This header file is included in the code generated for the Function Caller blocks associated with the function. The `fn1.h` file placement is not affected by the **Code Generation** > **Interface** > **Shared code placement** parameter value.

For more information, see “Generate Code for Functions and Callers” on page 4-6.

More About

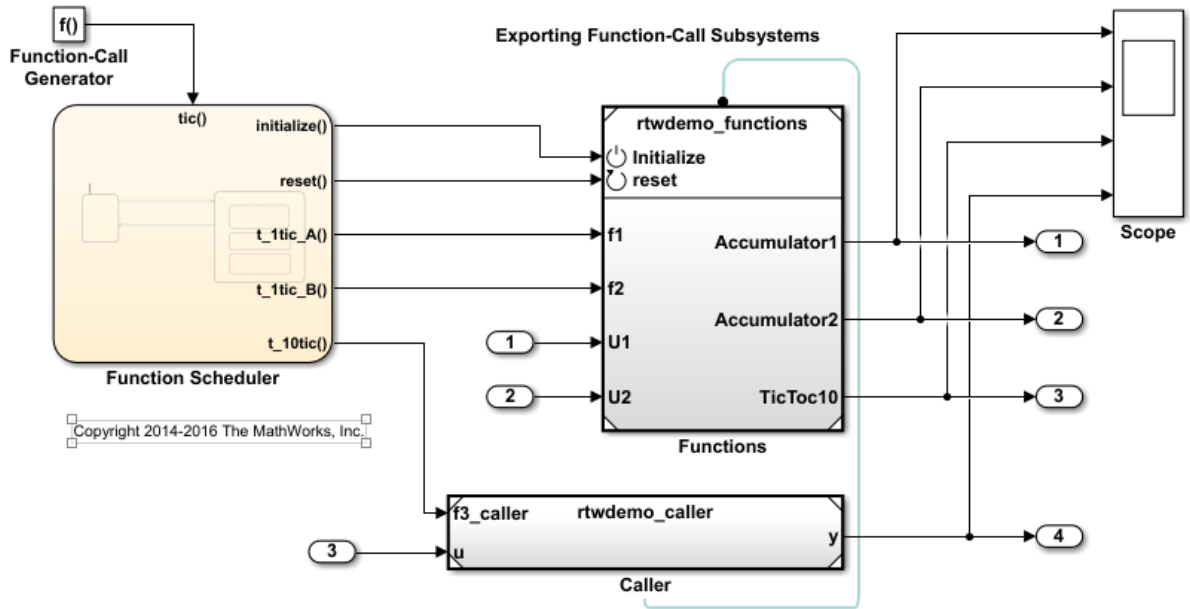
- “Design Models for Generated Embedded Code Deployment” on page 1-2
- “Generate Code for Functions and Callers” on page 4-6
- “Entry-Point Functions and Scheduling” on page 25-2
- “Configure AUTOSAR Client-Server Communication”
- “Simulink Functions in Models” (Simulink)
- “Simulink Functions in Referenced Models” (Simulink)
- “Simulink Functions” (Simulink)

Generate Code for Functions and Callers

In this section...

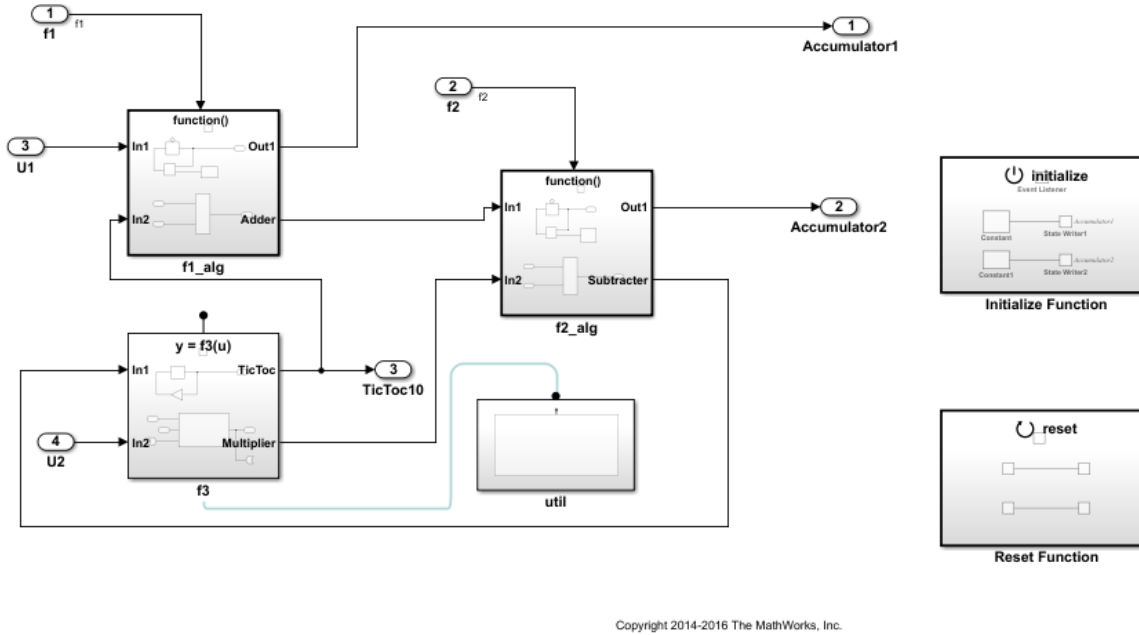
This example shows how the code generator translates Simulink Function blocks and Function Caller blocks into C code.

At the command prompt, type `rtwdemo_export_functions`. This model uses Stateflow software, but this example reviews only the code generated from the referenced models.



Generate Code for the Function Definition

1 Double click `rtwdemo_functions`. The Simulink Function block is the `f3` subsystem defined as `y = f3(u)`.



2 In the model window, press **Ctrl+B**.

The code generator creates `rtwdemo_functions.c`. This file contains the function definition and initialization.

- The initialization function is:

```
void f3_Init(void)
{
  /* InitializeConditions for UnitDelay: '<S5>/Delay' */
  rtDWork.Delay_DSTATE = 1;
}
```

- The primary function is:

```
/* Output and update for Simulink Function: '<Root>/f3' */
void f3(real_T rtu_u, real_T *rty_y)
{
  /* Outport: '<Root>/TicToc10' incorporates:
   *  UnitDelay: '<S5>/Delay'
```

```
   */
  rtY.TicToc10 = rtDWork.Delay_DSTATE;

  /* Gain: '<S5>/Gain' */
  rtDWork.Delay_DSTATE = (int8_T)(int32_T)-(int32_T)rtY.TicToc10;

  /* FunctionCaller: '<S5>/Function Caller' incorporates:
   *  Inport: '<Root>/U2'
   *  SignalConversion: '<S5>/TmpLatchAtIn1Outport1'
   *  SignalConversion: '<S5>/TmpSignal ConversionAtuOutport1'
   */
  adder_h(rtB.Subtract, rtU.U2, rtu_u, rtB.FunctionCaller);

  /* SignalConversion: '<S5>/TmpSignal ConversionAtyInport1' */
  *rty_y = rtB.FunctionCaller;
}
/* Output and update for Simulink Function: '<S6>/adder' */
void adder_h(real_T rtu_u1, real_T rtu_u2, real_T rtu_u3, real_T *rty_y)
{
  /* SignalConversion: '<S7>/TmpSignal ConversionAtyInport1' incorporates:
   *  SignalConversion: '<S7>/TmpSignal ConversionAtu1Outport1'
   *  SignalConversion: '<S7>/TmpSignal ConversionAtu2Outport1'
   *  SignalConversion: '<S7>/TmpSignal ConversionAtu3Outport1'
   *  Sum: '<S7>/Sum'
   */
  *rty_y = (rtu_u1 + rtu_u2) + rtu_u3;
}
```

- The shared header file, `f3.h`, contains the primary function prototype declaration.

```
/* Shared type includes */
#include "rtwtypes.h"

extern void f3(real_T rtu_u, real_T *rty_y);
```

Generate Code for the Caller Definition

1. On the `rtwdemo_export_functions` model, click `rtwdemo_caller`.
2. Press **Ctrl+B**.

The code generator creates the files `rtwdemo_caller.h` and `rtwdemo_caller.c` in the folder `rtwdemo_caller_ert_rtw`.

`rtwdemo_caller.h` includes the shared header file, `f3.h`, which contains the function prototype declaration.

`rtwdemo_caller.c` calls the function `f3`.

```
/* Output function for RootInportFunctionCallGenerator: '
        <Root>/RootFcnCall_InsertedFor_t_1Otic_at_outport_1' */
void rtwdemo_caller_t_1Otic(const real_T *rtu_u, real_T *rty_y)
{
  /* RootInportFunctionCallGenerator: '
        <Root>/RootFcnCall_InsertedFor_t_1Otic_at_outport_1' incorporates:
   *  SubSystem: '<Root>/Subsystem'
   */
  /* FunctionCaller: '<S1>/Function Caller' */
  f3(*rtu_u, rty_y);
}
```

More About

- "Design Models for Generated Embedded Code Deployment" on page 1-2
- "Modeling Functions and Callers for Code Generation" on page 4-2
- "Entry-Point Functions and Scheduling" on page 25-2
- "Configure AUTOSAR Client-Server Communication"

5

Referenced Models in Simulink Coder

Code Generation of Referenced Models

This section describes model referencing considerations that apply specifically to code generation by the Simulink Coder. This section assumes that you understand referenced models and related terminology and requirements, as described in "Overview of Model Referencing" (Simulink) and associated topics.

When generating code for a referenced model hierarchy, the code generator produces a stand-alone executable for the top model, and a library module called a *model reference target* for each referenced model. When the code executes, the top executable invokes the model reference targets to compute the referenced model outputs. Model reference targets are sometimes called *Simulink Coder targets*.

Be careful not to confuse a model reference target (Simulink Coder target) with other types of targets:

- Target hardware — A platform for which the Simulink Coder software generates code
- System target — A file that tells the Simulink Coder software how to generate code for particular purpose
- Rapid Simulation target (RSim) — A system target file supplied with the Simulink Coder product
- Simulation target — A MEX-file that implements a referenced model that executes with Simulink Accelerator™ software

The code generator places the code for the top model of a hierarchy in the code generation folder (Simulink) and places the code for referenced models in an `slprj` folder in the code generation folder (Simulink). Subfolders in `slprj` provide separate places for different types of files. For folder information, see "Manage Build Process Folders" (Simulink Coder).

By default, the product uses *incremental code generation*. When generating code, it compares structural checksums of referenced model files with the generated code files to determine whether to regenerate model reference targets. To control when rebuilds occur, use the configuration parameter **Model Referencing > Rebuild**. For details, see "Rebuild" (Simulink).

In addition to incremental code generation, the Simulink Coder software uses *incremental loading*. The code for a referenced model is not loaded into memory until the code for its parent model executes and needs the outputs of the referenced model. The

product then loads the referenced model target and executes. Once loaded, the target remains in memory until it is no longer used.

Most code generation considerations are the same whether or not a model includes referenced models: the code generator handles the details automatically insofar as possible. This chapter describes topics that you may need to consider when generating code for a model reference hierarchy.

If you have a Embedded Coder license, custom targets must declare themselves to be model reference compliant if they need to support Model blocks. For more information, see “Support Model Referencing” on page 71-83.

Generate Code for Referenced Models

In this section...

About Generating Code for Referenced Models

To generate code for referenced models, you

1. Create a subsystem in an existing model.
2. Convert the subsystem to a referenced model (Model block).
3. Call the referenced model from the top model.
4. Generate code for the top model and referenced model.
5. Explore the generated code and the code generation folder.

You can accomplish some of these tasks automatically with a function called `Simulink.Subsystem.convertToModelReference`.

Create and Configure the Subsystem

In the first part of this example, you define a subsystem for the `vdp` example model, set configuration parameters for the model, and use the `Simulink.Subsystem.convertToModelReference` function to convert it into two new models — the top model (`vdptop`) and a referenced model `vdpmultRM` containing a subsystem you created (`vdpmult`).

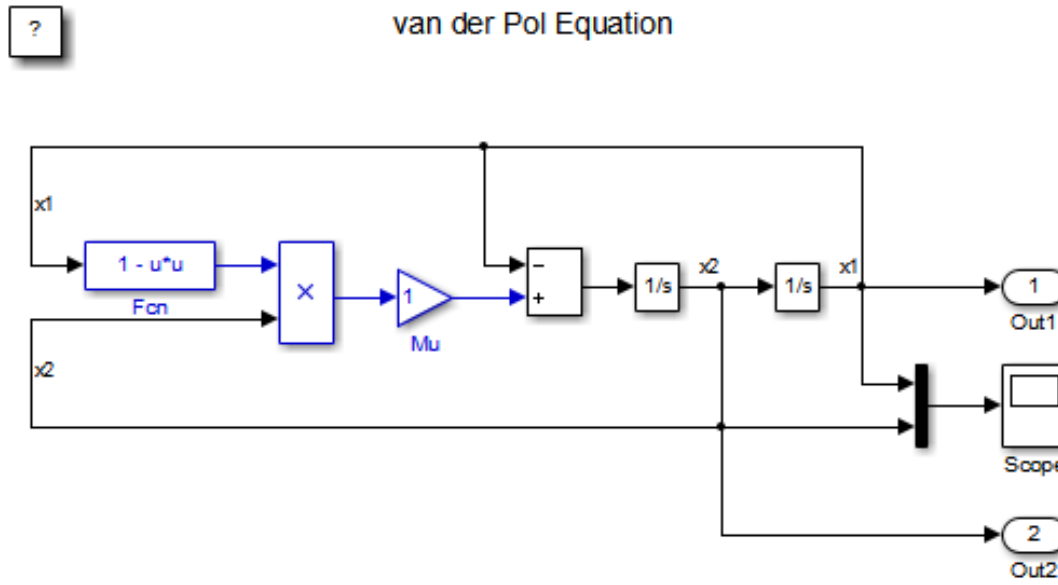
1. In the MATLAB Command Window, create a new working folder wherever you want to work and `cd` into it:

   ```
   mkdir mrexample
   cd mrexample
   ```

2. Open the `vdp` example model by typing:

`vdp`

3 Drag a box around the three blocks outlined in blue below:



4 Choose **Create Subsystem from Selection** from the **Diagram > Subsystem & Model Reference** menu.

A subsystem block replaces the selected blocks.

5 If the new subsystem block is not where you want it, move it to a preferred location.

6 Rename the block `vdpmult`.

7 Right-click the `vdpmult` block and select **Block Parameters (Subsystem)**.

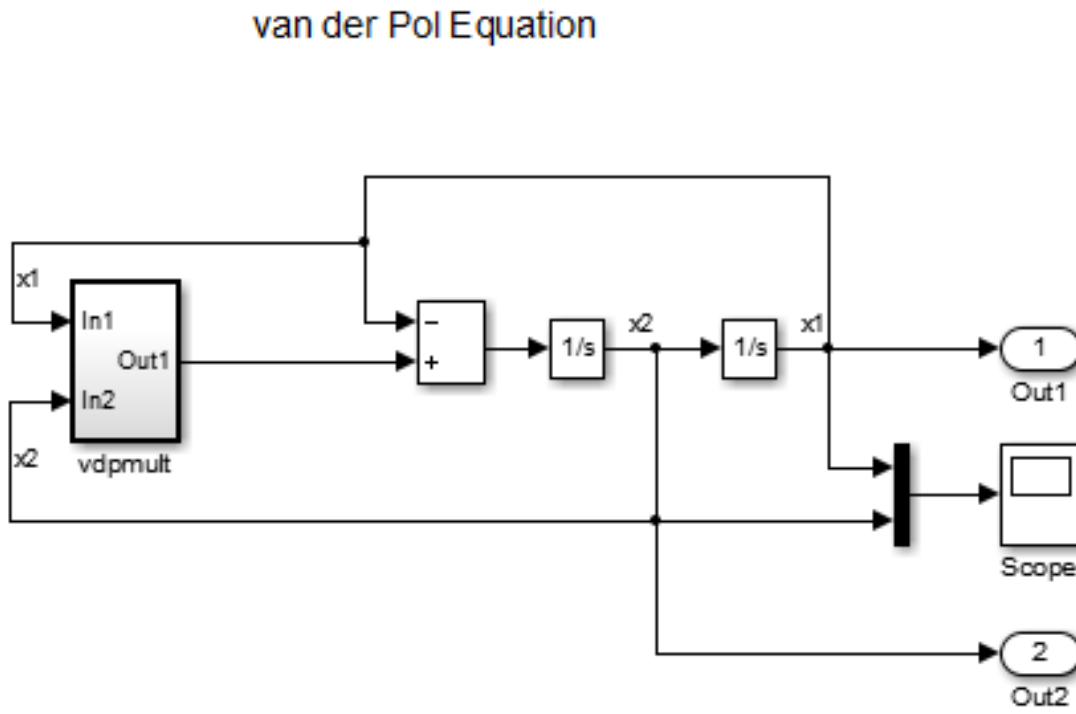
The **Function Block Parameters** dialog box appears.

8 In the **Function Block Parameters** dialog box, select **Treat as atomic unit**, then click **OK**.

The border of the `vdpmult` subsystem thickens to indicate that it is now atomic. An atomic subsystem executes as a unit relative to the parent model: subsystem block execution does not interleave with parent block execution. This property makes it

possible to extract subsystems for use as stand-alone models and as functions in generated code.

The block diagram should now appear as follows:



You must set several properties before you can extract a subsystem for use as a referenced model. To set the properties,

1. Open Model Explorer by selecting **Model Explorer** from the model's **View** menu.
2. In the **Model Hierarchy** pane, click the symbol preceding the model name to reveal its components.
3. Click **Configuration (Active)** in the left pane.
4. In the center pane, select **Solver**.
5. In the right pane, under **Solver Options** change the **Type** to `Fixed-step`, then click **Apply**. You must use fixed-step solvers when generating code, although referenced models can use different solvers than top models.

6 In the center pane, select **Diagnostics**. In the right pane, select the **Data Validity** tab. In the **Signals** area, set **Signal resolution** to `Explicit only`. Alternatively, if you do not want to use `Simulink.Signal` objects, set **Signal resolution** to `None`.

7 Click **Apply**.

The model now has the properties that model referencing requires.

8 In the center pane, click **Model Referencing**. In the right pane, set **Rebuild** to `If any changes in known dependencies detected`. Click **Apply**. This setting prevents code regeneration when it is not required.

9 In the `vdp` model window, choose **File** > **Save as**. Save the model as `vdptop` in your working folder. Leave the model open.

Convert Model to Use Model Referencing

In this portion of the example, you use the conversion function `Simulink.SubSystem.convertToModelReference` to extract the subsystem `vdpmult` from `vdptop` and convert `vdpmult` into a referenced model named `vdpmultRM`. To see the complete syntax of the conversion function, type at the MATLAB prompt:

```
help Simulink.SubSystem.convertToModelReference
```

For additional information, type:

```
doc Simulink.SubSystem.convertToModelReference
```

If you want to see an example of `Simulink.SubSystem.convertToModelReference` before using it yourself, type:

```
sldemo_mdlref_conversion
```

Simulink also provides a menu command, **Subsystem & Model Reference** > **Convert Subsystem to** > **Referenced Model**, that you can use to convert a subsystem to a referenced model. The command calls `Simulink.SubSystem.convertToModelReference` with default arguments. For more information, see "Convert a Subsystem to a Referenced Model" (Simulink).

Extract the Subsystem to a Referenced Model

To use `Simulink.SubSystem.convertToModelReference` to extract `vdpmult` and convert it to a referenced model, type:

```
Simulink.SubSystem.convertToModelReference...
('vdptop/vdpmult', 'vdpmultRM',...
'ReplaceSubsystem', true, 'BuildTarget', 'Sim')
```

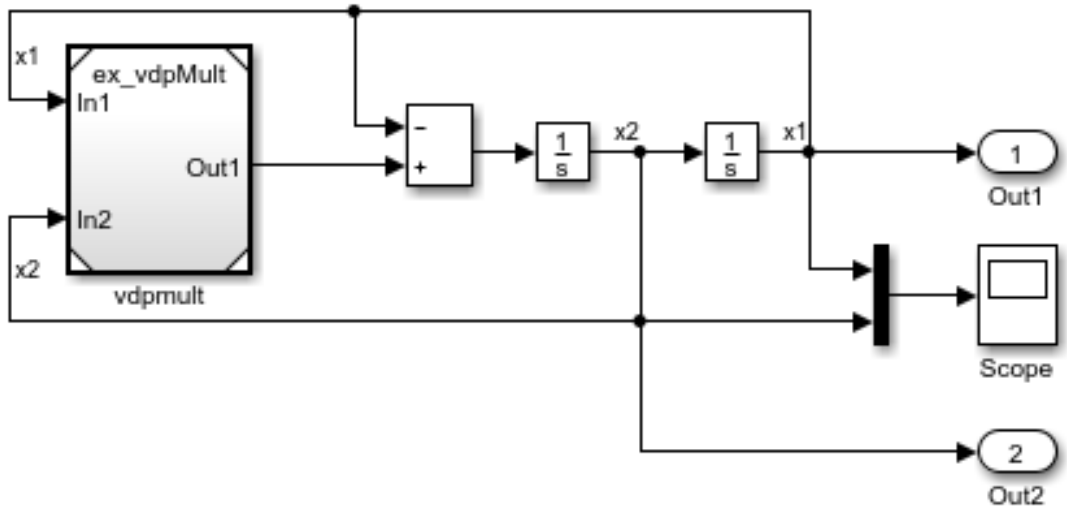
This command:

1. Extracts the subsystem `vdpmult` from `vdptop`.
2. Converts the extracted subsystem to a separate model named `vdpmultRM` and saves the model to the working folder.
3. In `vdptop`, replaces the extracted subsystem with a Model block that references `vdpmultRM`.
4. Creates a simulation target for `vdptop` and `vdpmultRM`.

The converter prints progress messages and terminates with

```
ans =
     1
```

The parent model `vdptop` now looks like this:

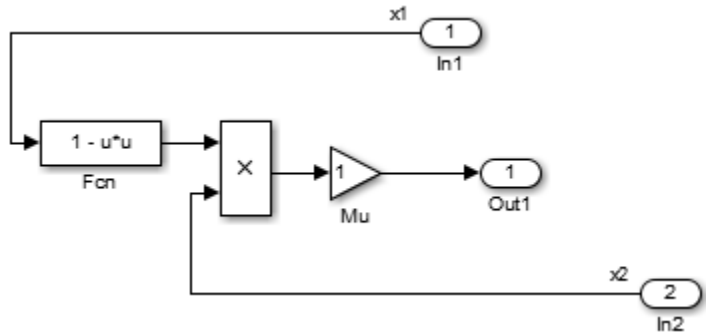


Note the changes in the appearance of the block `vdpmult`. These changes indicate that it is now a Model block rather than a subsystem. As a Model block, it does not have

contents of its own: the previous contents now exist in the referenced model `vdpmultRM`, whose name appears at the top of the Model block. Widen the Model block to expose the complete name of the referenced model.

If the parent model `vdptop` had been closed at the time of conversion, the converter would have opened it. Extracting a subsystem to a referenced model does *not* automatically create or change a saved copy of the parent model. To preserve the changes to the parent model, save `vdptop`.

Right-click the Model block `vdpmultRM` and choose **Open** to open the referenced model. The model looks like this:



Files Created and Changed by the Converter

The files in your working folder now consist of the following (not in this order).

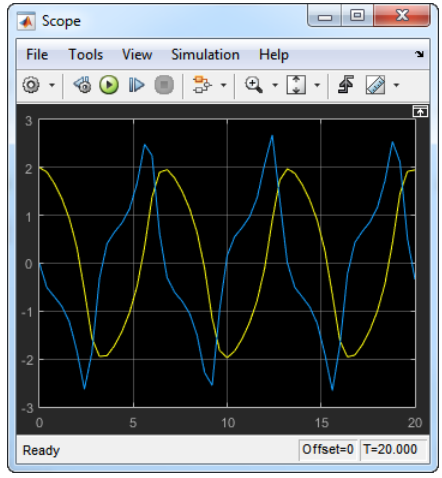
File	Description
`vdptop` model file	Top model that contains a Model block where the `vdpmult` subsystem was
`vdpmultRM` model file	Referenced model created for the `vdpmult` subsystem
`vdpmultRM_msf.mexw64`	Static library file (Microsoft® Windows platforms only). The file extension is system-dependent and may differ. This file executes when the `vdptop` model calls the Model block `vdpmult`. When called, `vdpmult` in turn calls the referenced model `vdpmultRM`.

File	Description
`/slprj`	Folder for generated model reference code

Code for model reference simulation targets is placed in the `slprj/sim` subfolder. Generated code for GRT, ERT, and other Simulink Coder targets is placed in `slprj` subfolders named for those targets. You will inspect some model reference code later in this example. For more information on code generation folders, see "Work with Code Generation Folders" on page 5-12.

Run the Converted Model

Open the Scope block in `vdptop` if it is not visible. In the `vdptop` window, click the **Run** tool or choose **Run** from the **Simulation** menu. The model calls the `vdpmultRM_msf` simulation target to simulate. The output looks like this:



Generate Model Reference Code for a GRT Target

The function `Simulink.SubSystem.convertToModelReference` created the model and the simulation target files for the referenced model `vdpmultRM`. In this part of the example, you generate code for that model and the `vdptop` model, and run the executable you create:

1. Verify that you are still working in the `mrexample` folder.
2. If the model `vdptop` is not open, open it. Make sure it is the active window.

3. Open Model Explorer by selecting **Model Explorer** from the model's **View** menu.
4. In the **Model Hierarchy** pane, click the symbol preceding the `vdptop` model to reveal its components.
5. Click **Configuration (Active)** in the left pane.
6. In the center pane, select **Data Import/Export**.
7. In the pane, select **Time** and **Output** and *clear* **Data stores**. Click **Apply**.

   These settings instruct the model `vdptop` (and later its executable) to log time and output data to MAT-files for each time step.
8. Generate GRT code (the default) and an executable for the top model and the referenced model. For example, in the model, press **Ctrl+B**.

The Simulink Coder build process generates and compiles code. The current folder now contains a new file and a new folder:

File	Description
`vdptop.exe`	The executable created by the build process
`vdptop_grt_rtw/`	The build folder, containing generated code for the top model

The build process also generated GRT code for the referenced model and placed it in the `slprj` folder.

To view a model's generated code in **Model Explorer**, the model must be open. To use the **Model Explorer** to inspect the newly created build folder, `vdptop_grt_rtw`:

1. Open Model Explorer by selecting **Model Explorer** from the model's **View** menu.
2. In the **Model Hierarchy** pane, click the symbol preceding the model name to reveal its components.
3. Click the symbol preceding `Code for vdptop` to reveal its components.
4. Directly under `Code for vdptop`, click `This Model`.

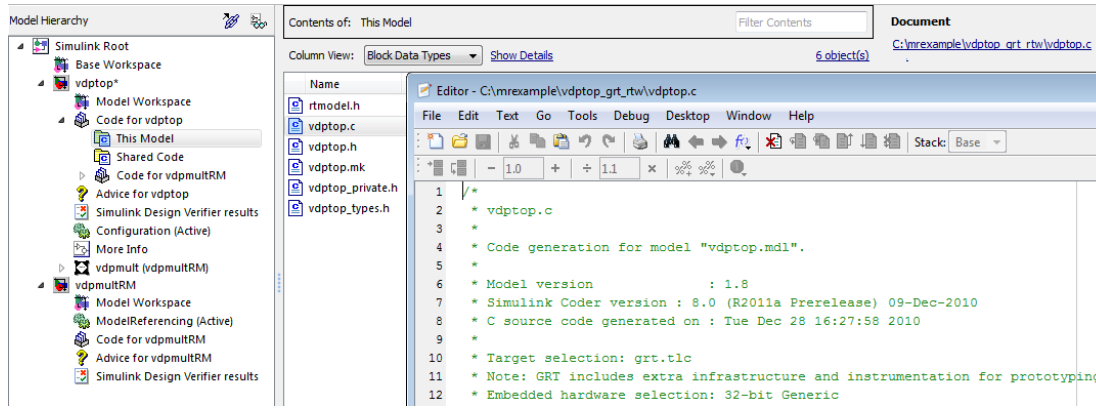
   A list of generated code files for `vdptop` appears in the **Contents** pane:

   ```
   rtmodel.h
   vdptop.c
   vdptop.h
   vdptop.mk
   ```

```
vdptop_private.h
vdptop_types.h
```

You can browse code by selecting a file of interest in the **Contents** pane.

To open a file in a text editor, click a filename, and then click the hyperlink that appears in the gray area at the top of the **Document** pane. The figure below illustrates viewing code for `vdptop.c`, in a text editor. Your code may differ.



To view the generated code in the HTML code generation report, see “Generate a Code Generation Report” (Simulink Coder).

Work with Code Generation Folders

When you view generated code in **Model Explorer**, the files listed in the **Contents** pane can exist either in a build folder or a code generation folder. Model reference code generation folders (located under the `slprj` folder), like build folders, are created in your code generation folder (Simulink). This process implies certain constraints on when and where model reference targets are built and on how they are accessed.

The models referenced by Model blocks can be stored anywhere. A given top model can include models stored on different file systems or in different folders. The same is not true for the simulation targets derived from these models; under most circumstances, models referenced by a given top model must be set up to simulate and generate model reference target code in a single code generation folder. The top and referenced models can exist anywhere on your path, but the code generation folder is assumed to exist in your current folder.

This means that, if you reference the same model from several top models, each stored in a different folder, you must either

- Always work in the same folder and be sure that the models are on your path
- Allow separate code generation folders, simulation targets, and Simulink Coder targets to be generated in each folder in which you work

The files in such multiple code generation folders are generally quite redundant. Therefore, to minimize code regeneration of referenced models, choose a specific working folder and remain in it for all sessions.

As model reference code generated for Simulink Coder targets as well as for simulation targets is placed in code generation folders, the same considerations as above apply even if you are generating target applications only. That is, code for all models referenced from a given model ends up being generated in the same code generation folder, even if it is generated for different targets and at different times.

Related Examples

- “Specify Instance-Specific Parameter Values for Reusable Referenced Model” on page 19-65

Configure Referenced Models

Minimize occurrences of algebraic loops by selecting the **Minimize algebraic loop occurrences** parameter on the **Model Reference** pane. The setting of this option affects only generation of code from the model. For information on how this option affects code generation, see "Configure Run-Time Environment Options" (Simulink Coder). For more information, see "Model Blocks and Direct Feed through" (Simulink).

Use the **Integer rounding mode** parameter on your model's blocks to simulate the rounding behavior of the C compiler that you intend to use to compile code generated from the model. This setting appears on the **Signal Attributes** pane of the parameter dialog boxes of blocks that can perform signed integer arithmetic, such as the Product and n-D Lookup Table blocks.

For most blocks, the value of **Integer rounding mode** completely defines rounding behavior. For blocks that support fixed-point data and the Simplest rounding mode, the value of **Signed integer division rounds to** also affects rounding. For details, see "Precision" (Fixed-Point Designer).

When models contain Model blocks, all models that they reference must be configured to use identical hardware settings. For information on the **Model Referencing** pane options, see "Model Configuration Parameters: Model Referencing" (Simulink) and "Configuration Parameter Requirements" (Simulink).

Build Model Reference Targets

By default, the Simulink engine rebuilds simulation targets before the Simulink Coder software generates model reference targets. You can change the rebuild criteria or specify when the engine rebuilds targets. For more information, see "Rebuild" (Simulink).

The Simulink Coder software generates a model reference target directly from the Simulink model. The product automatically generates or regenerates model reference targets, for example, when they require an update.

You can command the Simulink and Simulink Coder products to generate a simulation target for an Accelerator mode referenced model, and a model reference target for a referenced model, by executing the `slbuild` command with arguments in the MATLAB Command Window.

The Simulink Coder software generates only one model reference target for all instances of a referenced model. See "Reusable Code and Referenced Models" on page 5-26 for details.

Reduce Change Checking Time

You can reduce the time that the Simulink and Simulink Coder products spend checking whether simulation targets and model reference targets need to be rebuilt by setting configuration parameter values as follows:

- In the top model, consider setting the model configuration parameter **Model Referencing** > **Rebuild** to `If any changes in known dependencies detected`. (See "Rebuild" (Simulink).)
- In all referenced models throughout the hierarchy, set the configuration parameter **Diagnostics** > **Data Validity** > **Signal resolution** to `Explicit only` or `None`. (See "Signal resolution" (Simulink).)

These parameter values exist in a referenced model's configuration set, not in the individual Model block. Setting either value for an instance of a referenced model, sets it for all instances of that model.

Simulink Coder Model Referencing Requirements

A model reference hierarchy must satisfy various Simulink Coder requirements, as described in this section. In addition to these requirements, a model referencing hierarchy to be processed by the Simulink Coder software must satisfy:

- The Simulink requirements listed in:
  - "Configuration Requirements for All Referenced Model Simulation" (Simulink)
  - "Model Structure Requirements" (Simulink)
- The Simulink limitations listed in "Limitations on All Model Referencing" (Simulink)
- The Simulink Coder limitations listed in "Simulink Coder Model Referencing Limitations" on page 5-30

Configuration Parameter Requirements

A referenced model uses a configuration set in the same way a top model does, as described in "Manage a Configuration Set" (Simulink). By default, every model in a hierarchy has its own configuration set, which it uses in the same way that it would if the model executed independently.

Because each model can have its own configuration set, configuration parameter values can be different in different models. Furthermore, some parameter values are intrinsically incompatible with model referencing. The response of the Simulink Coder software to an inconsistent or unusable configuration parameter depends on the parameter:

- Where an inconsistency has no significance, the product ignores or resolves the inconsistency without posting a warning.
- Where a nontrivial and possibly acceptable solution exists, the product resolves the conflict silently; resolves it with a warning; or generates an error.
- If an acceptable resolution is not possible, the product generates an error. You must then change parameter values to eliminate the problem.

When a model reference hierarchy contains many referenced models that have incompatible parameter values, or a changed parameter value must propagate to many referenced models, manually eliminating all configuration parameter incompatibilities can be tedious. You can control or eliminate such overhead by using configuration

references to assign an externally-stored configuration set to multiple models. See "Manage a Configuration Reference" (Simulink) for details.

The following tables list configuration parameters that can cause problems if set in certain ways, or if set differently in a referenced model than in a parent model. Where possible, the Simulink Coder software resolves violations of these requirements automatically, but most cases require changes to the parameters in some or all models.

Configuration Requirements for Model Referencing with All System Targets

Dialog Box Pane	Option	Requirement
Solver	**Start time**	Some system targets require the start time of all models to be zero.
Hardware Implementation	All options	Values must be the same for top and referenced models.
Code Generation	**System target file**	Must be the same for top and referenced models.
	Language	Must be the same for top and referenced models.
	Generate code only	Must be the same for top and referenced models.
Symbols	**Maximum identifier length**	Cannot be longer for a referenced model than for its parent model.
Interface	**Code replacement library**	Must be the same for top and referenced models.
	C API options	The C API check boxes must be the same for top and referenced models.
	ASAP2 interface	Can be `on` or `off` in a top model, but must be `off` in a referenced model. If it is not, the Simulink Coder software temporarily sets it to `off` during code generation.

Configuration Requirements for Model Referencing with ERT System Targets (Requires Embedded Coder License)

Dialog Box Pane	Option	Requirement
All Parameters tab	**Ignore custom storage classes**	Must be the same for top and referenced models.
Symbols	**Global variables** **Global types** **Subsystem methods** **Local temporary variables** **Constant macros**	`$R` token must appear.
	Signal naming	Must be the same for top and referenced models.
	M-function	If specified, must be the same for top and referenced models.
	Parameter naming	Must be the same for top and referenced models.
	#define naming	Must be the same for top and referenced models.
Interface	**Support floating-point numbers**	Must be the same for both top and referenced models
	Support non-finite numbers	If `off` for top model, can be `off` or `off` for referenced models. If `on` for top model, must be `on` for referenced models.
	Support complex numbers	If `off` for top model, can be `off` or `off` for referenced models. If `on` for top model, must be `on` for referenced models.
	Suppress error status in real-time model	If `on` for top model, must be `on` for referenced models.
Code Placement	**Use owner from data object for data definition placement**	Must be the same for top and referenced models.

Dialog Box Pane	Option	Requirement
	Signal display level	Must be the same for top and referenced models.
	Parameter tune level	Must be the same for top and referenced models.

Naming Requirements

Within a model that uses model referencing, names of the constituent models can not collide. When you generate code from a model that uses model referencing, the **Maximum identifier length** parameter must be large enough to accommodate the root model name and the name-mangling text. A code generation error occurs if **Maximum identifier length** is not large enough.

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

Embedded Coder Naming Requirements

The Embedded Coder product lets you control the formatting of generated symbols in much greater detail. When generating code with an ERT target 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) except for **Shared utilities**.
- The **Maximum identifier length** must be large enough to accommodate full expansions of the `$R` and `$M` tokens.

See “Model Configuration Parameters: Code Generation Symbols” (Simulink Coder) for more information.

Custom Target Requirements

If you have an Embedded Coder license, a custom target must meet various requirements to support model referencing. For details, see “Support Model Referencing” on page 71-83.

Storage Classes for Signals Used with Model Blocks

Models containing Model blocks can use signals of storage class `Auto` without restriction. However, when you declare signals to be global, you must be aware of how the signal data will be handled.

A global signal is a signal with a storage class other than `Auto`:

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

The above are distinct from `SimulinkGlobal` signals, which are treated as test points with `Auto` storage class.

Global signals are declared, defined, and used as follows:

- An `extern` declaration is generated by all models that use a given global signal.

  As a result, if a signal crosses a Model block boundary, the top model and the referenced model both generate `extern` declarations for the signal.
- For an exported signal, the top model is responsible for defining (allocating memory for) the signal, whether or not the top model itself uses the signal.
- Global signals used by a referenced model are accessed directly (as global memory). They are not passed as arguments to the functions that are generated for the referenced models.

Custom storage classes also follow the above rules. However, certain custom storage classes are not currently supported for use with model reference. For details, see “Custom Storage Class Limitations” on page 23-71.

Storage Classes for Parameters Used with Model Blocks

Storage classes are supported for both simulation and code generation, and all except `Auto` are tunable. The supported storage classes thus include

- `SimulinkGlobal`
- `ExportedGlobal`

- `ImportedExtern`
- `ImportedExternPointer`
- `Custom`

Note the following restrictions on parameters in referenced models:

- Tunable parameters are not supported for noninlined S-functions.
- Tunable parameters set using the Model Parameter Configuration dialog box are ignored.

Note the following considerations concerning how global tunable parameters are declared, defined, and used in code generated for targets:

- A global tunable parameter is a parameter in the base workspace with a storage class other than `Auto`.
- An `extern` declaration is generated by all models that use a given parameter.
- If a parameter is exported, the top model is responsible for defining (allocating memory for) the parameter (whether it uses the parameter or not).
- Global parameters are accessed directly (as global memory). They are not passed as arguments to the functions that are generated for the referenced models.
- Symbols for `SimulinkGlobal` parameters in referenced models are generated using unstructured variables (`rtP_xxx`) instead of being written into the *`model`*`_P` structure. This is so that each referenced model can be compiled independently.

Certain custom storage classes for parameters are not currently supported for model reference. For details, see “Custom Storage Class Limitations” on page 23-71.

Parameters used as Model block arguments must be defined in the referenced model's workspace. For details, see “Parameterize Instances of a Reusable Referenced Model” (Simulink).

Signal Name Mismatches Across Model Reference Boundary

Within a parent model, the name and storage class for a signal entering or leaving a Model block might not match those of the signal attached to the root inport or outport within that referenced model. Because referenced models are compiled independently without regard to a parent model, they cannot adapt to the possible variations in how parent models label and store signals.

The Simulink Coder software accepts all cases where input and output signals in a referenced model have `Auto` storage class. When such signals are test pointed or are global, as described above, certain restrictions apply. The following table describes how mismatches in signal labels and storage classes between parent and referenced models are handled:

Relationships of Signals and Storage Classes Across Model Reference Boundary

Referenced Model	Parent Model	Signal Passing Method	Signal Mismatch Checking
`Auto`	Any storage class	Function argument	None
`SimulinkGlobal` or resolved to Signal Object	Any storage class	Function argument	**Signal label mismatch**
`Global`	Auto or `SimulinkGlobal`	Global variable	**Signal label mismatch**
`Global`	`Global`	Global variable	Labels and storage classes must be identical (else error)

To summarize, the following signal resolution rules apply to code generation:

- If the storage class of a root input or output signal in a referenced model is `Auto` (or is `SimulinkGlobal`), the signal is passed as a function argument.
  - When such a signal is `SimulinkGlobal` or resolves to a `Simulink.Signal` object, the **Signal label mismatch** diagnostic is applied.
- If a root input or output signal in a referenced model is global, it is communicated by using direct memory access (global variable). In addition,
  - If the corresponding signal in the parent model is also global, the names and storage classes must match exactly.
  - If the corresponding signal in the parent model is not global, the **Signal label mismatch** diagnostic is applied.

You can set the **Signal label mismatch** diagnostic to `error`, `warning`, or `none` in the **Diagnostics > Connectivity** pane of the Configuration Parameters dialog box.

Inherited Sample Time for Referenced Models

For information about Model block sample time inheritance, see "Sample Times for Model Referencing" (Simulink). In generated code, you can control inheriting sample time by using `ssSetModelReferenceSampleTimeInheritanceRule` in different ways:

- An S-function that precludes inheritance: If the sample time is used in the S-function's run-time algorithm, then the S-function precludes a model from inheriting a sample time. For example, consider the following `mdlOutputs` code:

  ```
  static void mdlOutputs(SimStruct *S, int_T tid)
  {
      const real_T *u = (const real_T*)
      ssGetInputPortSignal(S,0);
      real_T       *y = ssGetOutputPortSignal(S,0);
      y[0] = ssGetSampleTime(S,tid) * u[0];
  }
  ```

  This `mdlOutputs` code uses the sample time in its algorithm, and the S-function therefore should specify

  ```
  ssSetModelReferenceSampleTimeInheritanceRule
  (S, DISALLOW_SAMPLE_TIME_INHERITANCE);
  ```

- An S-function that does not preclude Inheritance: If the sample time is only used for determining whether the S-function has a sample hit, then it does not preclude the model from inheriting a sample time. For example, consider the `mdlOutputs` code from the S-function example `sfun_multirate.c`:

  ```
  static void mdlOutputs(SimStruct *S, int_T tid)
  {
      InputRealPtrsType enablePtrs;
      int               *enabled = ssGetIWork(S);

      if (ssGetInputPortSampleTime
      (S,ENABLE_IPORT)==CONTINUOUS_SAMPLE_TIME &&
      ssGetInputPortOffsetTime(S,ENABLE_IPORT)==0.0) {
          if (ssIsMajorTimeStep(S) &&
          ssIsContinuousTask(S,tid)) {
              enablePtrs =
              ssGetInputPortRealSignalPtrs(S,ENABLE_IPORT);
              *enabled = (*enablePtrs[0] > 0.0);
          }
      } else {
  ```

```
            int enableTid =
            ssGetInputPortSampleTimeIndex(S,ENABLE_IPORT);
            if (ssIsSampleHit(S, enableTid, tid)) {
                enablePtrs =
                ssGetInputPortRealSignalPtrs(S,ENABLE_IPORT);
                *enabled = (*enablePtrs[0] > 0.0);
            }
        }

        if (*enabled) {
            InputRealPtrsType uPtrs  =
            ssGetInputPortRealSignalPtrs(S,SIGNAL_IPORT);
            real_T            signal = *uPtrs[0];
            int               i;

            for (i = 0; i < NOUTPUTS; i++) {
                if (ssIsSampleHit(S,
                ssGetOutputPortSampleTimeIndex(S,i), tid)) {
                    real_T *y = ssGetOutputPortRealSignal(S,i);
                    *y = signal;
                }
            }
        }
    } /* end mdlOutputs */
```

The above code uses the sample times of the block, but only for determining whether there is a hit. Therefore, this S-function should set

```
ssSetModelReferenceSampleTimeInheritanceRule
(S, USE_DEFAULT_FOR_DISCRETE_INHERITANCE);
```

Customize Library File Suffix and File Type

You can control the library file suffix and file type extension that the Simulink Coder code generator uses to name generated model reference libraries. Use the model configuration parameter `TargetLibSuffix` to specify the scheme for the suffix and extension. The scheme must include a period (.). If you do not set this parameter, the Simulink Coder software names the libraries as follows:

- On Windows systems, *`model`*`_rtwlib.lib`
- On UNIX or Linux® systems, *`model`*`_rtwlib.a`

For more information, see "Using TargetLibSuffix with the Toolchain Approach" (Simulink Coder).

Reusable Code and Referenced Models

Models that employ model referencing might require special treatment when generating and using reusable code. The following sections identify general restrictions and discuss how reusable functions with inputs or outputs connected to a referenced model's root Inport or Outport blocks can affect code reuse.

General Considerations

You can generate code for subsystems that contain referenced models using the same procedures and options described in "Code Generation of Subsystems" (Simulink Coder). However, the following restrictions apply to such builds:

- A top model that uses single-tasking mode and that has a referenced model that uses multi-tasking mode executes for blocks with the different rates that are not connected. However, you get an error if the blocks with different rates are connected by Rate Transition block (inserted either manually or by Simulink).
- ERT S-functions do not support subsystems that contain a continuous sample time.
- The Simulink Coder S-function target is not supported.
- The Tunable parameters table (set by using the Model Parameter Configuration dialog box) is ignored; to make parameters tunable, you must define them as Simulink parameter objects in the base workspace.
- All other parameters are inlined into the generated code and S-function.

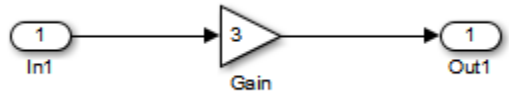
Note You can generate subsystem code using any target configuration available in the System Target File Browser. However, if the S-function target is selected, **Build This Subsystem** and **Build Selected Subsystem** behaves identically to **Generate S-Function**. (See "Automate S-Function Generation with S-Function Builder" on page 11-61.)

Code Reuse and Model Blocks with Root Inport or Outport Blocks

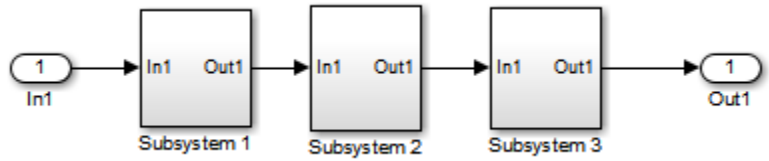
Reusable functions with inputs or outputs connected to a referenced model's root Inport or Outport block can affect code reuse. This means that code for certain atomic subsystems cannot be reused in a model reference context the same way it is reused in a standalone model.

For example, suppose you create the following subsystem and make the following changes to the subsystem's block parameters:

- Select **Treat as an atomic unit**
- Go to the **Code Generation** tab and set **Function packaging** to `Reusable function`



Suppose you then create the following model, which includes three instances of the preceding subsystem.



With the configuration parameter **Default parameter behavior** set to `Inlined` in this stand-alone model, the code generator can optimize the code by generating a single copy of the function for the reused subsystem, as shown below.

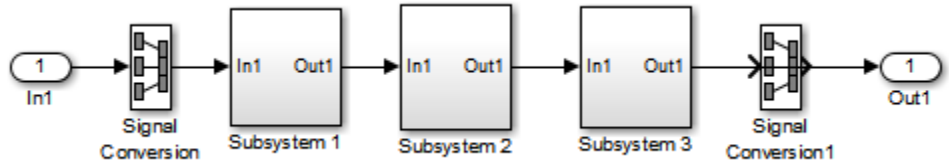
```
void reuse_subsys1_Subsystem1(
                   real_T rtu_0,
                   rtB_reuse_subsys1_Subsystem1 *localB)
{

  /* Gain: '<S1>/Gain' */
  localB->Gain_k = rtu_0 * 3.0;
}
```

When generated as code for a Model block (into an `slprj` folder in the code generation folder (Simulink)), the subsystems have three different function signatures:

```
/* Output and update for atomic system: '<Root>/Subsystem1' */
void reuse_subsys1_Subsystem1(const real_T *rtu_0,
```

```
rtB_reuse_subsys1_Subsystem1
 *localB)
{
  /* Gain: '<S1>/Gain' */
  localB->Gain_w = (*rtu_0) * 3.0;
}

/* Output and update for atomic system: '<Root>/Subsystem2' */
void reuse_subsys1_Subsystem2(real_T rtu_In1,
rtB_reuse_subsys1_Subsystem2
 *localB)
{
  /* Gain: '<S2>/Gain' */
  localB->Gain_y = rtu_In1 * 3.0;
}

/* Output and update for atomic system: '<Root>/Subsystem3' */
void reuse_subsys1_Subsystem3(real_T rtu_In1, real_T *rty_0)
{
  /* Gain: '<S3>/Gain' */
  (*rty_0) = rtu_In1 * 3.0;
}
```

One way to make all the function signatures the same for code reuse, is to insert Signal Conversion blocks. Place one between the Inport and Subsystem1 and another between Subsystem3 and the Outport of the referenced model.



The result is a single reusable function:

```
void reuse_subsys2_Subsystem1(real_T rtu_In1,
                  rtB_reuse_subsys2_Subsystem1 *localB)
{

  /* Gain: '<S1>/Gain' */
  localB->Gain_g = rtu_In1 * 3.0;
}
```

You can achieve the same result (reusable code) with only one Signal Conversion block. You can omit the Signal Conversion block connected to the Inport block if you select the **Pass fixed-size scalar root inputs by value** check box at the bottom of the **Model Referencing** pane of the Configuration Parameters dialog box. When you do this, you still need to insert a Signal Conversion block before the Outport block.

Simulink Coder Model Referencing Limitations

The following Simulink Coder limitations apply to model referencing. In addition to these limitations, a model reference hierarchy used for code generation must satisfy:

- The Simulink requirements listed in:
  - "Configuration Requirements for All Referenced Model Simulation" (Simulink)
  - "Model Structure Requirements" (Simulink)
- The Simulink limitations listed in "Model Referencing Limitations" (Simulink).
- The Simulink Coder requirements applicable to the code generation target, as listed in "Configuration Parameter Requirements" on page 5-16.

Customization Limitations

- The code generator ignores custom code settings in the Configuration Parameters dialog box and custom code blocks when generating code for a referenced model.
- Data type replacement is not supported for simulation target code generation of referenced models.
- Simulation targets do not include Stateflow target custom code.
- If you have an Embedded Coder license, some restrictions exist on grouped custom storage classes in referenced models. For details, see "Custom Storage Class Limitations" on page 23-71.

Data Logging Limitations

- To Workspace blocks, Scope blocks, and all types of runtime display, such as the display of port values and signal values, are ignored when the Simulink Coder software generates code for a referenced model. The resulting code is the same as if the constructs did not exist.
- Code generated for referenced models cannot log data to MAT-files. If data logging is enabled for a referenced model, the Simulink Coder software disables the option before code generation and re-enables it afterwards.
- If you log states for a model that contains referenced models, the ordering of the states in the output is determined by block sorted order, and might not match between simulation output and generated code MAT-file logging output.

State Initialization Limitation

When a top model uses the **Data Import/Export > Initial state** parameter in the Configuration Parameters dialog box to specify initial conditions, the Simulink Coder software does not initialize the discrete states of the referenced models during code generation.

Reusability Limitations

If a referenced model used for code generation has any of the following properties, the model must specify the configuration parameter **Model Referencing > Total number of instances allowed per top model** as `One`, and no other instances of the model can exist in the hierarchy. If you do not set the parameter to `One`, or more than one instance of the model exists in the hierarchy, an error occurs. The properties are:

- The model references another model which has been set to single instance
- The model contains a state or signal with non-auto storage class
- The model uses any of the following Stateflow constructs:
  - Machine-parented data
  - Machine-parented events
  - Stateflow graphical functions
- The model contains a subsystem that is marked as function
- The model contains an S-function that is:
  - Inlined but has not set the option `SS_OPTION_WORKS_WITH_CODE_REUSE`
  - Not inlined
- The model contains a function-call subsystem that:
  - Has been forced by the Simulink engine to be a function
  - Is called by a wide signal

For more information about **Total number of instances allowed per top model**, see "Total number of instances allowed per top model" (Simulink).

S-Function Limitations

- If a referenced model contains an S-function that should be inlined using a Target Language Compiler file, the S-function must use the `ssSetOptions` macro to set the `SS_OPTION_USE_TLC_WITH_ACCELERATOR` option in its `mdlInitializeSizes` method. The simulation target will not inline the S-function unless this flag is set.
- A referenced model cannot use noninlined S-functions generated by the Simulink Coder software.
- The Simulink Coder S-function target does not support model referencing.

For additional information, see "S-Functions with Referenced Models" (Simulink).

Simulink Tool Limitations

- Simulink tools that require access to a model's internal data or configuration (including the Model Coverage tool, the Simulink Report Generator product, the Simulink debugger, and the Simulink profiler) have no effect on code generated by the Simulink Coder software for a referenced model, or on the execution of that code. Specifications made and actions taken by such tools are ignored and effectively do not exist.

Subsystem Limitations

- If a subsystem contains Model blocks, you cannot build a subsystem module by right-clicking the subsystem (or by using **Code > C/C++ Code > Build Selected Subsystem**) unless the model is configured to use an ERT target.
- If you generate code for an atomic subsystem as a reusable function, inputs or outputs that connect the subsystem to a referenced model might prevent code reuse, as described in "Reusable Code and Referenced Models" (Simulink Coder).

Target Limitations

- The Simulink Coder S-function target does not support model referencing.

Other Limitations

- Errors or unexpected behavior can occur if a Model block is part of a cycle, the Model block is a direct feedthrough block, and an algebraic loop results. For details, see "Model Blocks and Direct Feed through" (Simulink).

- The **External mode** option is not supported. If it is enabled, it is ignored during code generation.
- When a model contains a trigger or enable port, you cannot generate standalone Simulink Coder code or PIL code.

6

Combined Models in Simulink Coder

Combine Code Generated for Multiple Models

Techniques

Techniques that you can use to combine code, which the code generator produces for multiple models or multiple instances of a model, into one executable program include:

- Referenced models. See "Overview of Model Referencing" (Simulink) and "Generate Code for Referenced Models" on page 5-4.
- If you have Embedded Coder software, interface the code for multiple models to a common harness program. From the harness program, call the entry-point functions generated for each model. The `ert.tlc` system target file has restrictions, relating to embedded processing, that could be incompatible with your application.
- Generate reusable, multi-instance code that is reentrant. See "Combine Code Generated for Multiple Models or Multiple Instances of a Model" on page 6-3.

The S-function system target (`rtwsfcn.tlc`) does not support combining code generated for multiple models.

Consider using model referencing to combine models for simulation and code generation. Model referencing helps with:

- Symbol naming consistency
- Required scheduling of the overall algorithm
- Model configuration consistency

If you combine code generated for different models (that is, without using referenced models), consider:

- Data is global. Symbol (name) clashes can result.
- Configuration parameter settings for the models must match, including settings such as hardware word sizes.
- Reuse and sharing of code can be suboptimal (for example, duplicate code for shared utility functions, scheduling, and solvers).
- Scheduling can be more complex (for example, models can have periodic sample times that are not multiples of each other, making scheduling from a common timer interrupt more complicated)

- For plant models that use continuous time and state, the continuous time signals connecting models are not handled by a single solver like continuous time signals within a model. This can lead to subtle numeric differences.

Control Ownership of Data

If you have Embedded Coder software, you can specify an *owner* for individual data items such as signals, parameters, and states. The owner of a data item generates the definition (memory allocation and initialization) for the data item. For example, if you apply a custom storage class to a `Simulink.Parameter` object so that it appears as a tunable global variable in the generated code, specify one of the combined models as the owner of the object. The code generated for that model defines the parameter data.

If you use model referencing, you can modularize the generated code and establish clear ownership of data when you work in a team.

If you do not use model referencing, you can prevent generation of duplicate definitions for a data item. For example, suppose you store a `Simulink.Parameter` object in the base workspace and apply the storage class `ExportedGlobal`. If you generate code from two separate models that use the object, each model generates a definition for the corresponding global variable. Instead, you can specify an owner for the object so that only the owner generates a definition.

To specify an owner for a data item:

1. Apply a custom storage class to the data item. See “Introduction to Custom Storage Classes” on page 23-2.
2. Configure the owner of the data item by specifying the **Owner** custom attribute.
3. Select the model configuration parameter **Use owner from data object for data definition placement**.

For more information about controlling ownership and file placement of data definitions and declarations, see “Manage Placement of Data Definitions and Declarations” on page 36-100.

Combine Code Generated for Multiple Models or Multiple Instances of a Model

For each model for which you are combining code, generate the code.

1. Set the system target file to a GRT- or ERT-based system target file. The system target file for the models you combine, must be the same.
2. If you intend to have multiple instances of that model in the application, set the model configuration parameter **Code Generation > Interface > Code interface packaging** to `Reusable function`. If you specified an ERT-based system target file, optionally, you can set the model configuration parameter **Use dynamic memory allocation for model initialization**, depending on whether you want to statically or dynamically allocate the memory for each instance of the model.
3. Generate source code. The code generator includes an allocation function in the generated file *`model`*`.c`. The allocation function dynamically allocates model data for each instance of the model.

After generating source code for each model:

1. Compile the code for each model that you are combining.
2. Combine the makefiles generated for the models into one makefile.
3. Create a combined simulation engine by modifying a main program, such as `rt_malloc_main.c`. The main program initializes and calls the code generated for each model.
4. Run the makefile. The makefile links the object files and the main program into an executable program.

Share Data Across Models

Use unidirectional signal connections between models. This affects the order in which models are called. For example, if you use an output signal from `modelA` as input to `modelB`, the `modelA` output computation should be called first.

Timing Issues

When combining code generated for multiple models or multiple instances of a model:

- Configure the models with the same solver mode (single-tasking or multitasking).
- If the models use continuous states, configure the models with the same solver.

If the base rates for the models differ, the main program (such as `rt_malloc_main.c`) must set up the timer interrupt to occur at the greatest common divisor rate of the models. The main program calls each model at a time interval.

Data Logging and External Mode Support

A multiple-model program can log data to separate MAT-files for each model.

Only one of the models in a multiple-model program can use external mode.

7

Configure Model Parameters for Simulink Coder

Configure Run-Time Environment Options

When you use Simulink software to create and execute a model and use the code generator to produce C or C++ code, consider your configuration for up to three run-time environments:

- The MATLAB development computer run-time environment that runs MathWorks software during application development.
- The production hardware run-time environment in which you deploy an application when it is put into production.
- The test hardware run-time environment in which you test an application under development before deployment.

One run-time environment can serve in multiple capacities, but the run-time environments remain conceptually distinct. Often, the MATLAB development computer is the test hardware. Typically, the production hardware is different from, and less powerful than, the MATLAB development or the test hardware. Many types of production hardware can do little more than run a downloaded executable file.

Provide information about the production hardware board and the compiler that you use with it when:

- You use Simulink software to simulate a model for which you later generate code
- You use the code generator to produce code for deployment on *production* hardware

The software uses the board and compiler information to get bit-true agreement for the results of integer and fixed-point operations performed in simulation and in code generated for the production hardware. The code generator uses the information to create code that executes with maximum efficiency.

When you generate code for testing on *test* hardware, provide information about the test hardware board and the compiler that you use. The code generator uses this information to create code that provides bit-true agreement between results from:

- Integer and fixed-point operations performed in simulation
- Generated code run on the production hardware
- Generated code run on the test hardware

You can achieve bit-true agreement for results even if the production and test hardware are different. Where the C standard does not completely define behavior, the compilers for the two types of hardware can use different defaults.

Configure Production and Test Hardware

You can specify model simulation or code generation for a specific hardware board and its device type. For example, you can set the data size, byte ordering, and compiler behavior, such as integer rounding. You can configure:

- The production hardware and the compiler that you use with it. This information affects simulation and code generation. See "Example Production Hardware Setting That Affects Normal Mode Simulation" on page 7-13.
- The test hardware and the compiler that you use with it. This information affects only code generation.

Configure production hardware by selecting **Configuration Parameters** > **Hardware Implementation**. By default, the Hardware Implementation pane lists **Hardware board**, **Device vendor**, and **Device type** parameter fields only. Unless you have installed hardware support packages, **Hardware board** lists values `None` or `Determine by Code Generation system target file`, and `Get Hardware Support Packages`. After installing a hardware support package, the list also includes the corresponding hardware board name. If you select a hardware board name, parameters for that board appear. To set device details, such as data size and byte ordering, click **Device details**.

Configure test hardware in the Configuration Parameters dialog box, on the **All Parameters** tab. To enable parameters for configuring test hardware details, set `ProdEqTarget` to `off`. Code generated for test hardware executes in the environment specified by the test hardware parameters. The code behaves as if it were executing in the environment specified for the production hardware. For more information, see "Test Hardware Considerations" on page 7-13.

Default values and properties appear as initial values in the **Hardware Implementation** pane when:

- You specify a **System target file** in the **Code Generation** pane.
- The system target file specifies a default microprocessor and its hardware properties.

You cannot change parameters that have only one possible value. Parameters that have more than one possible value provide a list of valid values. If you specify hardware properties manually in **Hardware Implementation** pane, verify that these values are consistent with the system target file. Otherwise, the generated code can fail to compile or execute, or can execute but produce incorrect results.

Hardware implementation parameters describe hardware and compiler properties to MATLAB software. The code generator uses the information to produce code for the run-time environment that runs as efficiently as possible. The generated code gives bit-true agreement for the results of integer and fixed-point operations in simulation, production code, and test code.

For details about specific parameters, see "Hardware Implementation Pane" (Simulink). To see an example of **Hardware Implementation** pane capabilities, see the `rtwdemo_targetsettings` example model. For details related to configuring a hardware implementation, see:

Specify Hardware Board

Specify the hardware board that runs the code generated from your model. Select a value for **Configuration Parameters > Hardware Implementation > Hardware board**.

The Hardware Implementation pane identifies the system target file selected on **Configuration Parameters > Code Generation**.

To configure test hardware, use **Configuration Parameters > All Parameters**.

To enable parameters for configuring test hardware details, set `ProdEqTarget` to `off`.

Ways to Specify the Hardware Board

If	Select
The menu *includes* the name of the hardware board that you want to use.	The name of that hardware board.

If	Select
	If you select a hardware board name, parameters for that board appear.
The menu *does not include* the name of the hardware board that you want to use.	`Get Hardware Support Packages`. That value opens the Support Package Installer. Install the support package that you want. After you install the support package, the menu includes relevant hardware board names.
The model configuration *uses* system target file `ert.tlc`, `realtime.tlc`, or `autosar.tlc`.	`None`. No hardware board is specified for the hardware implementation.
The model configuration *does not use* system target file `ert.tlc`, `realtime.tlc`, or `autosar.tlc`.	`Determine by Code Generation system target file`. The code generator uses the specified system target file to determine the hardware implementation.

Specify Device Vendor

To specify the vendor of the microprocessor of the hardware device, use the **Device vendor** parameter. Your selection determines the available microprocessors in the **Device type** menu. If the vendor name does not appear, select `Custom Processor`. Then, use the **Device type** parameter to specify the microprocessor.

- For complete lists of **Device vendor** and **Device type** values, see "Device vendor" (Simulink) and "Device type" (Simulink).
- To add **Device vendor** and **Device type** values to the default set that is displayed on the **Hardware Implementation** pane, see "Register More Device Vendor and Device Type Values" on page 7-6.

Specify the Device Type

To specify the microprocessor name from the supported devices listed for your **Device vendor** selection, use the **Device type** parameter. If the microprocessor does not appear in the menu, change **Device vendor** to `Custom Processor`. Then, specify device details for your custom device.

If you select a device type for which the system target file specifies default hardware properties, the properties appear as initial values. You cannot change the value of parameters with only one possible selection. Parameters that have more than one possible value provide a menu. Select values for your hardware.

Register More Device Vendor and Device Type Values

To add **Device vendor** and **Device type** values to the default set that is displayed on the **Hardware Implementation** pane, you can use a hardware device registration API provided by the code generator.

To use this API, you create an `sl_customization.m` file, on your MATLAB path, that invokes the `registerTargetInfo` function and fills in a hardware device registry entry with device information. The device information is registered with Simulink software for each subsequent Simulink session. (To register your device information without restarting MATLAB, issue the MATLAB command `sl_refresh_customizations`.)

For example, the following `sl_customization.m` file adds device vendor `MyDevVendor` and device type `MyDevType` to the Simulink device lists.

```
function sl_customization(cm)
  cm.registerTargetInfo(@loc_register_device);
end

function thisDev = loc_register_device
  thisDev = RTW.HWDeviceRegistry;
  thisDev.Vendor = 'MyDevVendor';
  thisDev.Type = 'MyDevType';
  thisDev.Alias = {};
  thisDev.Platform = {'Prod', 'Target'};
  thisDev.setWordSizes([8 16 32 32 32]);
  thisDev.LargestAtomicInteger = 'Char';
  thisDev.LargestAtomicFloat = 'None';
  thisDev.Endianess = 'Unspecified';
  thisDev.IntDivRoundTo = 'Undefined';
  thisDev.ShiftRightIntArith = true;
  thisDev.setEnabled({'IntDivRoundTo'});
end
```

After device registration, you can select the device in the **Hardware Implementation** pane.

To register multiple devices, specify an array of `RTW.HWDeviceRegistry` objects in your `sl_customization.m` file. For example:

```
function sl_customization(cm)
  cm.registerTargetInfo(@loc_register_device);
end

function thisDev = loc_register_device

  thisDev(1) = RTW.HWDeviceRegistry;
  thisDev(1).Vendor = 'MyDevVendor';
  thisDev(1).Type = 'MyDevType1';
  ...

  thisDev(4) = RTW.HWDeviceRegistry;
  thisDev(4).Vendor = 'MyDevVendor';
  thisDev(4).Type = 'MyDevType4';
  ...

end
```

You can specify various `RTW.HWDeviceRegistry` properties in the `registerTargetInfo` function call in your `sl_customization.m` file.

Properties for registerTargetInfo Function Call

Property	Description
`Vendor`	Character vector specifying the **Device vendor** value for your hardware device.
`Type`	Character vector specifying the **Device type** value for your hardware device.
`Alias`	Cell array of character vectors specifying aliases or legacy names that can resolve to this device. Specify each alias or legacy name in the format `'Vendor->Type'`. Embedded Coder software provides the utility functions `RTW.isHWDeviceTypeEq` and `RTW.resolveHWDeviceType`. These functions detect and resolve alias values or legacy values when testing user-specified values for the hardware device type.
`Platform`	Cell array of enumerated character vector values specifying whether this device can be listed in the **Production hardware** subpane (`{'Prod'}`), the **Test hardware** subpane (`{'Target'}`), or both (`{'Prod', 'Target'}`).

Property	Description
`setWordSizes`	Array of integer sizes to associate with the **Number of bits** parameters **char**, **short**, **int**, **long**, and **native word size**, respectively.
`LargestAtomicInteger`	Character vector specifying an enumerated value for the **Largest atomic size: integer** parameter: `'Char'`, `'Short'`,`'Int'`, or `'Long'`.
`LargestAtomicFloat`	Character vector specifying an enumerated value for the **Largest atomic size: floating-point** parameter: `'Float'`, `'Double'`, or `'None'`.
`Endianess`	Character vector specifying an enumerated value for the **Byte ordering** parameter: `'Unspecified'`, `'Little'` for little Endian, or `'Big'` for big Endian.
`IntDivRoundTo`	Character vector specifying an enumerated value for the **Signed integer division rounds to** parameter: `'Zero'`, `'Floor'`, or `'Undefined'`.
`ShiftRightIntArith`	Boolean value specifying whether your compiler implements a signed integer right shift as an arithmetic right shift (`true`) or not (`false`).
`setEnabled`	Cell array of character vectors specifying which device properties you can modify in the **Hardware Implementation** pane when you select this device type. This property applies for the `'Endianess'`, `'IntDivRoundTo'`, and `'ShiftRightIntArith'` properties. You can apply this property to individual **Number of bits** parameters by using the property names `'BitPerChar'`, `'BitPerShort'`, `'BitPerInt'`, `'BitPerLong'`, and `'NativeWordSize'`.

Set Bit Lengths for Device Data Types

The **Number of bits** parameters describe the **native word size** of the microprocessor and the bit lengths of **char**, **short**, **int**, and **long** data. For code generation to succeed:

- The bit lengths must be such that **char** <= **short** <= **int** <= **long**.
- Bit lengths must be multiples of 8, with a maximum of 32.

- The bit length for **long** data must not be less than 32.

The `rtwtypes.h` file defines integer type names. The values that you provide must be consistent with the word sizes as defined in the compiler `limits.h` header file. The code generator maps its integer type names to the corresponding Simulink integer type names.

If no ANSI® C type with a matching word size is available, but a larger ANSI C type is available, the code generator uses the larger type for `int8_T`, `uint8_T`, `int16_T`, `uint16_T`, `int32_T`, and `uint32_T`. When the code generator uses a larger type, the resulting logged values (for example, MAT-file logging) can have different data types than logged values for simulation.

An application can use an integer data of length from 1 (unsigned) or 2 (signed) bits up 32 bits. If the integer length matches the length of an available type, the code generator uses that type. If a matching type is not available, the code generator uses the smallest available type that can hold the data, generating code that does not use unnecessary higher-order bits. For example, on hardware that supports 8-bit, 16-bit, and 32-bit integers, for a signal specified as 24 bits, the code generator implements the data as an `int32_T` or `uint32_T`.

Code that uses emulated integer data is not maximally efficient. This code can be useful during application development for emulating integer lengths that are available only on production hardware. Emulation does not affect the results of execution.

During code generation, the software checks the compatibility of model data types with the data types that you specify for production hardware.

- If none of the lengths that you specify for production hardware integers is 32 bits, the software generates an error.
- If the lengths of data types that the model uses are smaller than the available production hardware integer lengths, the software generates a warning.

Mapping of Integer Types from Code Generator to Simulink

Code Generator Integer Type	Simulink Integer Type
`boolean_T`	`boolean`
`int8_T`	`int8`
`uint8_T`	`uint8`
`int16_T`	`int16`

Code Generator Integer Type	Simulink Integer Type
`uint16_T`	`uint16`
`int32_T`	`int32`
`uint32_T`	`uint32`

Set Byte Ordering for Device

The **Byte ordering** parameter specifies whether the hardware uses `Big Endian` (most significant byte first) or `Little Endian` (least significant byte first) byte ordering. If left as `Unspecified`, the code generator produces code that determines the endianness of the hardware. This setting is the least efficient.

Set Quotient Rounding Behavior for Signed Integer Division

ANSI C does not completely define the quotient rounding technique for compilers to use when dividing one signed integer by another. So, the behavior is implementation-dependent. If both integers are positive, or both are negative, the quotient must round down. If either integer is positive and the other is negative, the quotient can round up or down.

The **Signed integer division rounds to** parameter instructs the code generator about how the compiler rounds the result of signed integer division. Providing this information does not change the operation of the compiler. It only describes that behavior to the code generator, which uses the information to optimize code generated for signed integer division. The parameter values are:

- `Zero` — If the quotient is between two integers, the compiler chooses the integer that is closer to zero as the result.
- `Floor` — If the quotient is between two integers, the compiler chooses the integer that is closer to negative infinity.
- `Undefined` — If `Zero` or `Floor` do not describe the compiler behavior or if that behavior is unknown, choose this value.

Avoid selecting `Undefined`. When the code generator does not know the signed integer division rounding behavior of the compiler, the model build generates extra code.

The compiler quotient rounding behavior varies according to these values.

You can obtain the compiler implementation for signed integer division rounding from the compiler documentation. If documentation is not available, you can determine this behavior by experiment.

Example Quotient Rounding for Zero, Floor, and Undefined

N	D	Ideal N/D	Zero	Floor	Undefined
33	4	8.25	8	8	8
-33	4	-8.25	-8	-9	-8 or -9
33	-4	-8.25	-8	-9	-8 or -9
-33	-4	8.25	8	8	8 or 9

Set Arithmetic Right Shift Behavior for Signed Integers

ANSI C does not define the behavior of right shifts on negative integers for compilers. So, the behavior is implementation-dependent. The **Shift right on a signed integer as arithmetic shift** option instructs the code generator about how the compiler implements right shifts on negative integers. Providing this information does not change the operation of the compiler. It only describes that behavior to the code generator, which uses the information to optimize the code generated for arithmetic right shifts.

If the C compiler implements a signed integer right shift as an arithmetic right shift, select the option. Otherwise, clear the option. An arithmetic right shift fills bits vacated by the right shift with the value of the most significant bit, which indicates the sign of the number in two's-complement notation. The option is selected by default. If your compiler handles right shifts as arithmetic shifts, this setting is preferred.

- When you select the option, the code generator produces efficient code whenever the Simulink model performs arithmetic shifts on signed integers.
- When the option is cleared, the code generator produces fully portable but less efficient code to implement right arithmetic shifts.

You can obtain the compiler implementation for arithmetic right shifts from the compiler documentation. If documentation is not available, you can determine this behavior by experiment.

Update Release 14 Hardware Configuration

If your model was created before Release 14 and you have not updated the model, the **Configure current execution hardware device** parameter (`TargetUnknown`) value is `'on'` by default.

To update your model, clear the box for **Configuration Parameters > All Parameters > Configure current execution hardware device**.Or in the Command Window, type:

```
cs = getActiveConfigSet('your_model_name');
set_param(cs,'TargetUnknown','off');
```

This update to your model:

- Enables the **Test Hardware is the same as production hardware** parameter (`ProdEqTarget`), setting the parameter to `'on'`.
- Copies the **Production device vendor and type** parameter (`ProdHWDeviceType`) value to the **Test device vendor and type** parameter (`TargetHWDeviceType`).

To complete the update:

1. Clear the box for **Configuration Parameters > All Parameters > Test Hardware is the same as production hardware**. Apply this step only if your production and test hardware are different.
2. Set the parameters in **Configuration Parameters > All Parameters > Hardware implementation** to match your production and test systems.
3. Save the model.

Production Hardware Considerations

When you configure production hardware, consider these points:

- Production hardware can have word sizes and other hardware characteristics that differ from the MATLAB development computer. You can prototype code on hardware that is different from the production hardware or the MATLAB development computer. When producing code, the code generator accounts for these differences.
- The Simulink product uses some of the information in the production hardware configuration. That information enables simulations without code generation to give the same results as executing generated code. For example, the results can detect error conditions that arise on the production hardware, such as hardware overflow.
- The code generator produces code that provides bit-true agreement with Simulink results for integer and fixed-point operations. Generated code that emulates unavailable data lengths runs less efficiently than without emulation. The emulation does not affect bit-true agreement with Simulink for integer and fixed-point results.
- If you change run-time environments during application development, before generating or regenerating code, reconfigure the hardware implementation parameters for the new run-time environment. When code executes on hardware for which it was not generated, bit-true agreement is not always achieved for results of integer and fixed-point operations in simulation, production code, and test code.

- To compile code generated from the model, use the **Integer rounding mode** parameter on model blocks to simulate the rounding behavior of the C compiler that you intend. This setting appears on the **Signal Attributes** pane in the parameter dialog boxes of blocks that can perform signed integer arithmetic, such as the Product and n-D Lookup Table blocks.
- For most blocks, the value of **Integer rounding mode** completely defines rounding behavior. For blocks that support fixed-point data and the simplest rounding mode, the value of **Signed integer division rounds to** also affects rounding. For details, see "Precision" (Fixed-Point Designer).
- When models contain Model blocks, configure models that they reference to use identical hardware settings.

Test Hardware Considerations

By default, the test hardware configuration is the same as the configuration for the production hardware. You can use the generated code for testing in an environment that is identical to the production environment.

If the test and production environments differ, you can generate code that runs on test hardware as if it were running on production hardware:

1. In the Configuration Parameters dialog box, on the **All Parameters** tab, enable test hardware parameters by setting `ProdEqTarget` to `off`.
2. Specify device type details through the test hardware (`Target*`) parameters.

If you select a system target file that specifies a default microprocessor and its hardware properties, these default values and properties appear as initial values.

Parameters with only one possible value cannot be changed. If you modify hardware properties, check that their values are consistent with the system target file. Otherwise, the generated code can fail to compile or execute, or can execute but produce incorrect results.

Example Production Hardware Setting That Affects Normal Mode Simulation

Changing some production hardware settings, for example, `ProdLongLongMode` and `ProdIntDivRoundTo`, can affect normal mode simulation results. The following example simulates an adder with four inputs. In the first simulation, `ProdLongLongMode` is disabled. In the second simulation, `ProdLongLongMode` is enabled. In the plot of

simulation outputs, you observe small differences between output values in the time step range 125–175.

```
model = 'hwSettingEffect';
new_system(model)
open_system(model)

% Create adder
pos = [140   140   200   340];
add_block('simulink/Math Operations/Add',[model '/sum_int32'], ...
          'Inputs','++++','SaturateOnIntegerOverflow','on','Position',pos)

pos = [75   155   105   175];
add_block('built-in/Inport',[model '/In1'],'Position',pos)
set_param([model '/In1'], 'OutDataTypeStr','int32','PortDimensions','1','SampleTime','1');
add_line(model, 'In1/1','sum_int32/1')

pos = [75   205   105   225];
add_block('built-in/Inport',[model '/In2'],'Position',pos)
set_param([model '/In2'], 'OutDataTypeStr','int32','PortDimensions','1','SampleTime','1');
add_line(model, 'In2/1','sum_int32/2')

pos = [75   255   105   275];
add_block('built-in/Inport',[model '/In3'],'Position',pos)
set_param([model '/In3'], 'OutDataTypeStr','int32','PortDimensions','1','SampleTime','1');
add_line(model, 'In3/1','sum_int32/3')

pos = [75   305   105   325];
add_block('built-in/Inport',[model '/In4'],'Position',pos)
set_param([model '/In4'], 'OutDataTypeStr','int32','PortDimensions','1','SampleTime','1');
add_line(model, 'In4/1','sum_int32/4')

pos = [275   230   305   250];
add_block('built-in/Outport',[model '/Out1'],'Position',pos)
add_line(model, 'sum_int32/1','Out1/1')

% Specify input data
t = 0:200;
peakValue = 1.5e9;
in1 = peakValue * sin(t*2*pi/100);
in2 = peakValue * cos(t*2*pi/70);
in3 = -peakValue * sin(t*2*pi/40);
in4 = -peakValue * cos(t*2*pi/30);
set = Simulink.SimulationData.Dataset;
set = set.addElement(1, timeseries(int32(in1),t,'Name','sig1'));
set = set.addElement(2, timeseries(int32(in2),t,'Name','sig2'));
set = set.addElement(3, timeseries(int32(in3),t,'Name','sig3'));
set = set.addElement(4, timeseries(int32(in4),t,'Name','sig4'));

set_param(model, 'LoadExternalInput', 'on');
set_param(model, 'ExternalInput', 'set');

set_param(model, 'StopTime', '50');

% Disable production hardware setting and run first simulation
```

```
set_param(model, 'ProdLongLongMode', 'off');
[~, ~, y1] = sim(model, 200);

% Enable production hardware setting and run second simulation
set_param(model, 'ProdLongLongMode', 'on');
[~, ~, y2] = sim(model, 200);

plot([y1 y2]);
figure(gcf);
```

The difference in behavior is due to the accumulator data type in the Sum block. The **Accumulator data type** block parameter is set to `Inherit: Inherit via internal rule`. For this example, the resulting accumulator data type is 64 bits wide if the use of the C `long long` data type is enabled. Otherwise, it is 32 bits wide. Depending on the input values for the sum block, the 32-bit accumulator can saturate when the 64-bit accumulator does not. Therefore, normal mode behavior can depend on the `ProdLongLongMode` setting. In both cases, the normal mode behavior and production hardware behavior matches bitwise.

More About

- “Hardware Implementation Pane” (Simulink)
- “Device vendor” (Simulink)
- “Device type” (Simulink)
- “Precision” (Fixed-Point Designer)

8

Model Protection in Simulink Coder

Protect a Referenced Model

Protect a model when you want to share a model with a third party without revealing intellectual property. Protecting a model does not use encryption technology unless you use the optional password protection available for read-only view, simulation, and code generation. If you choose password protection for one of these options, the software protects the supporting files using AES–256 encryption.

When you create a protected model (Simulink):

- By default, Simulink creates and stores a protected version of the referenced model in the current working folder. The protected model has the same name as the source model, with a `.slxp` extension.
- The original Model block does not change. However, if the Model block parameter **Model name** does not specify an extension, a protected model, `.slxp`, takes precedence over a model file, `.slx`.
- You can optionally create a harness model which includes the protected model. A shield icon appears in the lower-left corner of the protected model block in the harness model. For more information, see “Harness Model” on page 8-4.
- You can optionally include generated code with the protected model so that a third party can generate code for a model that contains the protected model. For more information, see “Code Generation Support in a Protected Model” on page 8-6.
- If the Model block uses variants, only the active variant is protected. For more information, see “Set up Model Variants” (Simulink).
- If the model defines callbacks, the model protection process does not preserve these callbacks. For more information on creating callbacks for use with a protected model, see “Define Callbacks for Protected Model” (Simulink Coder).
- If you rename a protected model, or change its suffix, the model is unusable until you restore its original name and suffix. You cannot change a protected model file internally because such changes make the file unusable.

Create a protected model using one of the following options.

- The Model block context menu. For more information, see “Create a Protected Model” on page 8-10
- The `Simulink.ModelReference.protect` function.
- The Simulink Editor menu bar. Select **File > Export Model To > Protected Model** to create a protected model from the current model.

Requirements for Protecting a Model

When you create a protected model from a referenced model, the referenced model must meet all requirements listed in "Model Referencing Limitations" (Simulink), as well as these requirements:

- You must have a Simulink Coder license to create a protected model.
- A model that you protect must be available on the MATLAB path and not have unsaved changes.
- A model that you protect cannot reference a protected model directly or indirectly.
- A model that you protect cannot use a non-inlined S-function directly or indirectly.
- To use a protected model that requires passwords across platforms, before you create the protected model, set the MATLAB character set encoding to `'US-ASCII'`. For more information, see `slCharacterEncoding`.

Model protection has certain limitations, as listed in "Limitations on All Model Referencing" (Simulink) and "Limitations on Accelerator Mode Referenced Models" (Simulink).

Harness Model

You can create a harness model for the generated protected model. The harness model opens as a new, untitled model that contains only a Model block that references the protected model. This Model block:

- Specifies the Model block parameter, **Model name**, as the name of the protected model.
- Has a shield icon in the lower-left corner.
- Has the same number of input and output ports as the protected model.
- Defines model reference arguments that the protected model uses, but does not provide values.

To create a harness model, see “Create a Protected Model” on page 8-10. You can use a harness model to test your protected model. For more information, see “Test the Protected Model” on page 8-20. You can also copy the Model block in your harness model to another model, where it is an interface to the protected model.

Protected Model Report

You can generate a protected model report when you create the protected model. The report is included as part of the protected model. The report has:

- A **Summary**, including the following tables:
  - **Environment**, providing the Simulink version and platform used to create the protected model.
  - **Supported functionality**, reporting `On`, `Off`, or `On with password protection` for each possible functionality that the protected model supports.
  - **Licenses**, listing licenses required to run the protected model.
- An **Interface Report**, including model interface information such as input and output specifications, exported function information, interface parameters, and data stores.

When you create the protected model from the Simulink Editor, the protected model report is generated. To generate a report when using the `Simulink.ModelReference.protect` function, set the '`Report`' option to `true`.

If you configure your protected model for multiple targets, the **Summary** includes a list of supported targets in the **Supported functionality** table. When you build a model that references a protected model with multiple targets, the protected model code generation report represents the currently configured target.

To view the protected model report, right-click the protected-model badge icon and select **Display Report**. Or, call the `Simulink.ProtectedModel.open` function with the `report` option.

Code Generation Support in a Protected Model

You can create a protected model that supports code generation. When a protected model includes generated code, a third party can generate code for a model that includes the protected model. If you choose to obfuscate the code, the code is obfuscated before compilation. The protected model file contains only obfuscated headers and binaries. Source code, such as `.c` and `.cpp`, is not present in the protected model file, although the headers are documented in the protected model report. For more information, see "Protected Model File" on page 8-8 and "Protected Model Report" on page 8-5.

In the Create Protected Model dialog box, select the **Use generated code** check box. The appearance of the generated code is determined by the **Content type** list. To enable obfuscated code, select `Obfuscated source code` from the list. For an example on including code generation support, see "Create a Protected Model" on page 8-10.

Protected Model Requirements to Support Code Generation

Contents and configuration of a model might prevent code generation support of the protected model. Interaction between the parent model and the protected model might also prevent code generation.

- Code generation for the protected model is only supported for Normal, Accelerator, Software-in-the-Loop (SIL), and Processor-in-the-Loop (PIL) modes and a single target.
- Source code comments in the **Code Generation > Comments** pane are ignored. Obfuscation of the generated code removes comments because comments might reveal intellectual property.
- Custom code specified in the **Code Generation > Custom Code** pane is obfuscated, but identifiers are not.
- Code generation of a model that includes a protected model causes an error, if:
  - Their interfaces do not match.
  - There are incompatible parameters.
  - A protected model and another model share the same name in the same model reference hierarchy.
- Selecting the **Code Generation > Verification > Measure function execution time** check box is incompatible with model protection. If you have this option selected

when you protect your model, the software turns the parameter off and displays a warning.

Protected Model File

A protected model file (`.slxp`) consists of the model itself and supporting files, depending on the options that you selected when you created the protected model.

If you created a protected model for simulation only and the referencing model is in `Normal` mode, after simulation, the *`model`*`.mexext` file is placed in the build folder.

If you created a protected model for simulation only and the referencing model is in `Accelerator` or `Rapid Accelerator` mode, after simulation, the following files are unpacked:

- `slprj/sim/`*`model`*`/*.h`
- `slprj/sim/`*`model`*`/`*`model`*`lib.a` (or *`model`*`lib.lib`)
- `slprj/sim/`*`model`*`/tmwinternal/*`
- `slprj/sim/_sharedutils/*`

For the protected model report, these additional files are unpacked (but not in the build folder):

- `slprj/sim/`*`model`*`/html/*`
- `slprj/sim/`*`model`*`/buildinfo.mat`

If you opted to include code generation support when you created the protected model, after building your model the following files are unpacked (in addition to the preceding files):

- `slprj/sim/`*`model`*`/*.h`
- `slprj/sim/`*`model`*`/`*`model`*`lib.a` (or *`model`*`lib.lib`)
- `slprj/sim/`*`model`*`/tmwinternal/*`
- `slprj/sim/_sharedutils/*`
- `slprj/`*`target`*`/`*`model`*`/*.h`
- `slprj/`*`target`*`/`*`model`*`/`*`model`*`_rtwlib.a` (or *`model`*`_rtwlib.lib`)
- `slprj/`*`target`*`/`*`model`*`/buildinfo.mat`
- `slprj/`*`target`*`/`*`model`*`/codeinfo.mat`
- `slprj/`*`target`*`/_sharedutils/*`
- `slprj/`*`target`*`/`*`model`*`/tmwinternal/*`

With an Embedded Coder license, you can specify a `Top model` code interface. In this case, if you opted to include code generation support when you created the protected model, after building your model the following files are unpacked:

- `slprj/sim/model/*.h`
- `slprj/sim/model/modellib.a` (or `modellib.lib`)
- `slprj/sim/model/tmwinternal/*`
- `slprj/sim/_sharedutils/*`
- `model_target_rtw/*.h`
- `model_target_rtw/*.objExt`
- `model_target_rtw/buildinfo.mat`
- `model_target_rtw/codeinfo.mat`
- `slprj/target/_sharedutils/*`
- `slprj/target/model/tmwinternal/*`

For the protected model report, after building your model these files are unpacked (in addition to the preceding files):

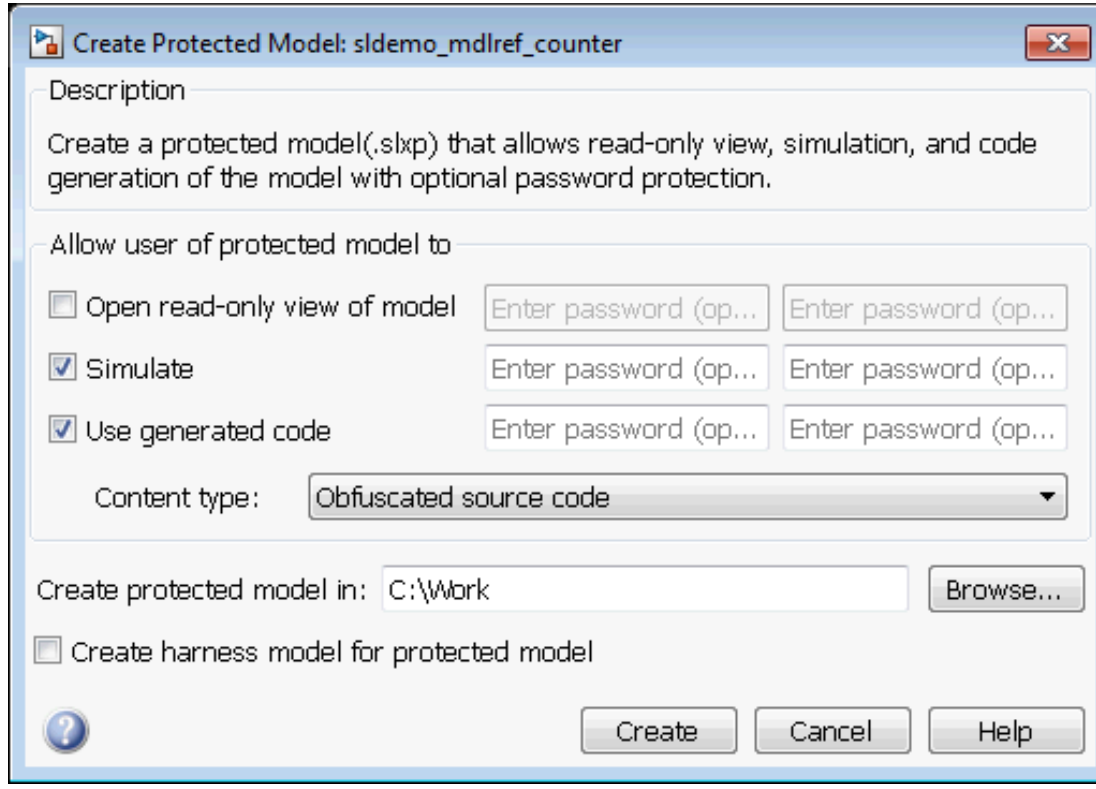
- `slprj/target/model/html/*`
- `slprj/target/model/buildinfo.mat`
- `slprj/target/_sharedutils/html/*`

Note: The `slprj/sim/model/*` files are deleted after they are used.

Create a Protected Model

This example shows how to create a protected model for read-only viewing, simulation, or code generation.

1. Open your model. For this example, `sldemo_mdlref_basic` is used as a demonstration.
2. In the Simulink Editor, right-click the model block that references the model for which you want to generate protected model code. In this example, right-click CounterA.
3. From the context menu, select **Block Parameters (ModelReference)**.
4. In the Block Parameters dialog box, in the **Model name** field, specify the extension for the model, `.slx`. When both the model and the protected model exist in the same folder, `.slxp` takes precedence over `.slx`. In the **Model name** field, if you do not specify an extension, then the original model block in the model becomes protected.
5. Click **OK**.
6. Right-click the model block. From the context menu, select **Subsystem & Model Reference > Create Protected Model for Selected Model Block**.



7 In the Create Protected Model dialog box, select the **Simulate** and **Use generated code** check boxes. If you want to password-protect the functionality of the protected model, enter a password with a minimum of four characters. Each functionality can have a unique password.

8 If you have an Embedded Coder license and specify an ERT based system target file (for example, `ert.tlc`) for the model, the **Code interface** field is visible. From the **Code interface** drop-down list, select one of the following options:

- `Model reference` — Specifies code access through the model reference code interface, which allows use of the protected model within a model reference hierarchy. Users of the protected model can generate code from a parent model that contains the protected model. In addition, users can run Model block SIL/PIL simulations with the protected model.

- `Top model` — Specifies code access through the standalone interface. Users of the protected model can run Model block SIL/PIL simulations with the protected model.

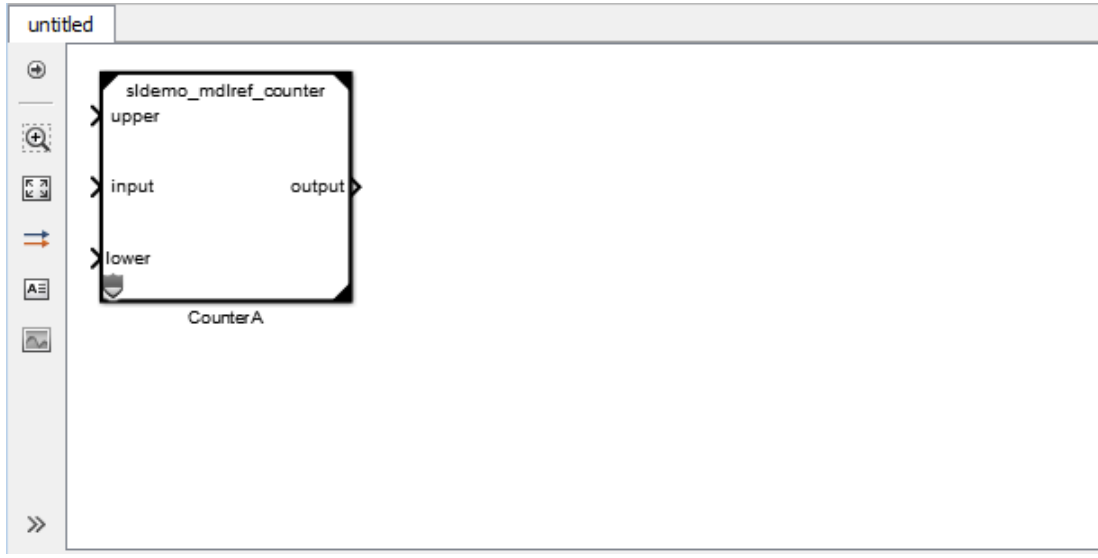
Note: In this example, `sldemo_mdlref_basic` does not specify an ERT based system target file, therefore the Code interface options are not available on the Create Protected Model dialog box.

9 From the **Content type** list, select `Obfuscated source code` to conceal the source code purpose and logic of the protected model.

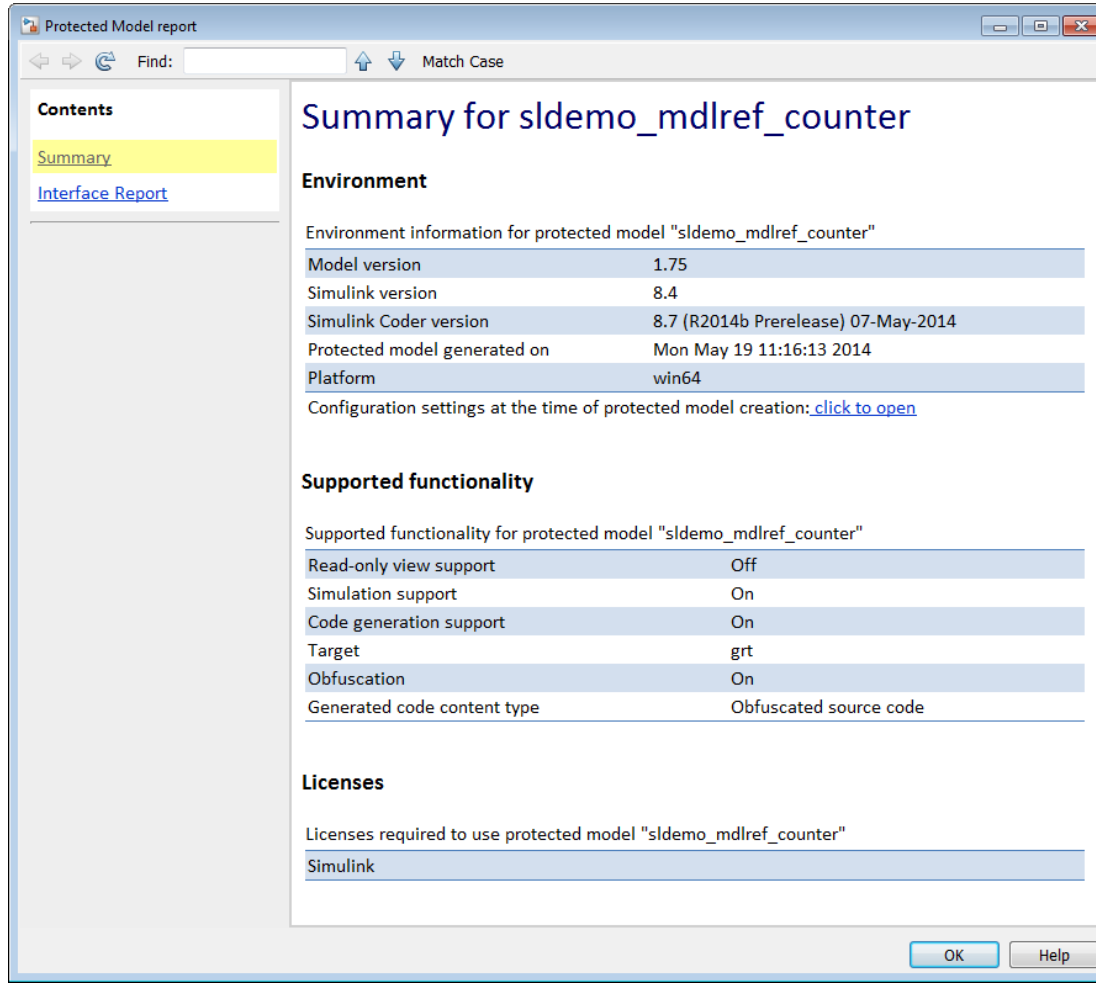
10 In the **Create protected model in** field, specify the folder path for the protected model. The default value is the current working folder.

11 To create a harness model for the protected model, select the **Create harness model for protected model** check box.

12 Click **Create**. An untitled harness model opens. It contains a model block, which refers to the protected model `sldemo_mdlref_counter.slxp`. The **Simulation mode** for the Model block is set to `Accelerator`. You cannot change the mode.



13 To view the protected model report, right-click the protected-model badge icon on the `CounterA` block and select **Display Report** .



Related Examples

- “Test the Protected Model” on page 8-20
- “Package a Protected Model” on page 8-23
- “Configure and Run SIL Simulation” on page 64-15

More About

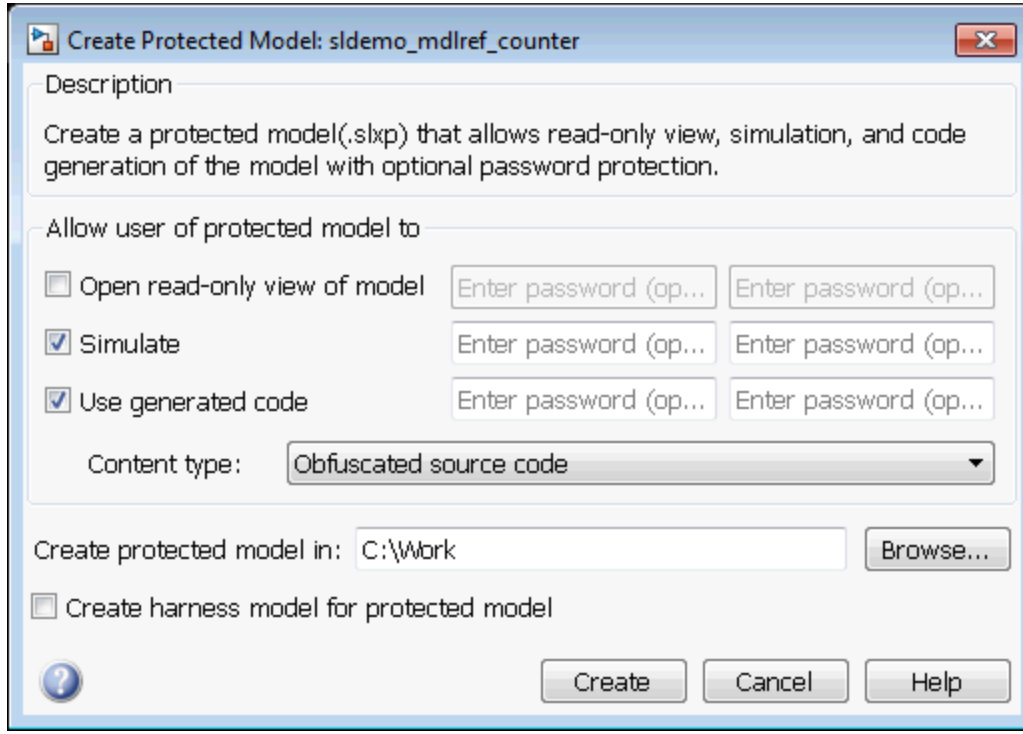
- "Code Generation Support in a Protected Model" on page 8-6
- "Protected Model Creation Settings" on page 8-15
- "Code Interfaces for SIL and PIL" on page 64-6

Protected Model Creation Settings

When you create a protected model, in the Create Protected Model dialog box, you can select which settings you want configured. The settings provide certain functionality permissions when using a protected model. The functionality choices are:

- Read-only viewing
- Simulation
- Code Generation

Password-protection is optional. You must have a minimum of four characters.



Open Read-Only View of Model

If you want to share a view-only version of your model, this option will allows someone using the protected model to open a Web view of the model. The contents and block parameters are viewable in the model Web view.

Simulate

The **Simulate** check box allows someone to simulate a protected model. When you select this check box, the Web view is not inherited. To enable the Web view with simulation functionality, select the **Open Read-Only View of Model** check box. The **Simulate** functionality:

- Enables protected model Simulation Report.
- Sets Mode to Accelerator. You can run Normal Mode and Accelerator simulations.
- Displays only binaries and headers.
- Enables code obfuscation.

Use Generated Code

The **Use generated code** check box allows simulation and code generation for a protected model. To generate code, the **Simulate** check box must also be selected. This functionality:

- Enables protected model Simulation Report and Code Generation Report.
- Sets Mode to enable code generation.
- Enables support for simulation.
- Supports the Model block if you have an Embedded Coder license and specify an ERT system target file (`ert.tlc`) for the model. From the **Code interface** drop-down list, select one of the following options:
  - `Model reference` — Specifies the model reference code interface, which allows use of the protected model within a model reference hierarchy. Users of the protected model can generate code from a parent model that contains the protected model. In addition, users can run Model block SIL/PIL simulations with the protected model.
  - `Top model` — Specifies code access through the standalone interface. Users of the protected model can run Model block SIL/PIL simulations with the protected model.

- Determines the appearance of the generated code by the **Content type** list. The options are:
  - `Binaries`
  - `Obfuscated source code` (default)
  - `Readable source code`, which also includes readable code comments

Create a Protected Model with Multiple Targets

You can create a protected model that supports multiple code generation targets. This example shows how to use command-line functions to create a protected model that supports code generation for GRT and ERT targets.

1. Load a model and save a local copy. This model is configured for a GRT target.

```
sldemo_mdlref_counter
save_system('sldemo_mdlref_counter','mdlref_counter.slx');
```

2. Add a required password for modifying a protected model. If you do not add a password, you are prompted to set a password when you create a modifiable, protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'mdlref_counter','password');
```

3. Create a modifiable, protected model with support for code generation.

```
Simulink.ModelReference.protect('mdlref_counter','Mode',...
'CodeGeneration', 'Modifiable',true, 'Report',true);
```

4. Get a list of targets that the protected model supports.

```
 st = Simulink.ProtectedModel.getSupportedTargets('mdlref_counter')
```

```
st =

    'grt'    'sim'
```

5. Configure the unprotected model to support an ERT target.

```
 set_param('mdlref_counter', 'SystemTargetFile', 'ert.tlc');
 save_system('mdlref_counter');
```

6. Add support to the protected model for the ERT target. You are prompted for the modification password.

```
 Simulink.ProtectedModel.addTarget('mdlref_counter');
```

7. Verify that the list of supported targets now includes the ERT target.

```
 st = Simulink.ProtectedModel.getSupportedTargets('mdlref_counter')
```

```
st =

    'ert'    'grt'    'sim'
```

Use a Protected Model with Multiple Targets

When using a protected model with multiple targets, prepare your model for code generation.

1. Get a list of the targets that the protected model supports using the `Simulink.ProtectedModel.getSupportedTargets` function.

   You can also get this information from the protected model report. To view the report, on the protected model block, right-click the badge icon. Select **Display Report**. The **Summary** lists the supported targets.

2. Get the configuration set for your chosen target using the `Simulink.ProtectedModel.getConfigSet` function. You can use the configuration set to verify that the protected model interface is compatible with the parent model.

3. Generate code. The build process selects the corresponding target.

Test the Protected Model

To test a protected model that you created, you use the generated harness model and the procedure described in "Use Protected Model in Simulation" (Simulink).

You can also compare the output of the protected model to the output of the original model. Because you are the supplier, both the original and the protected model might exist on the MATLAB path. In the original model, if the Model block **Model name** parameter names the model without providing a suffix, the protected model takes precedence over the unprotected model. If you need to override this default when testing the output, in the Model block **Model name** parameter, specify the file name with the extension of the unprotected model, `.slx`.

To compare the unprotected and protected versions of a Model block, use the Simulation Data Inspector. This example uses `sldemo_mdlref_basic` and the protected model, `sldemo_mdlref_counter.slxp`, which is created in "Create a Protected Model" on page 8-10.

1. If it is not already open, open `sldemo_mdlref_basic`.
2. Enable logging for the output signal of the Model block, `CounterA`. In the Configuration Parameters dialog box, in the **Data Import/Export** pane, select the **Signal logging** parameter. Click **Apply** and **OK**.
3. Right-click the output signal. From the context menu, select `Properties`. In the Signal Properties dialog box, select **Log signal data**. Click **Apply** and **OK**. For more information, see "Export Signal Data Using Signal Logging" (Simulink).
4. Right-click the `CounterA` block. From the context menu, select `Block Parameters (ModelReference)`. In the Block Parameters dialog box, specify the **Model name** parameter with the name of the unprotected model and the extension, `sldemo_mdlref_counter.slx`. Click **Apply** and **OK**. Repeat this for `CounterB` block and `CounterC` block.
5. In the Simulink Editor, click the **Simulation Data Inspector** button arrow and select **Send Logged Workspace Data to Data Inspector** from the menu.
6. Simulate the model. When the simulation is complete, click the **Simulation Data Inspector** button to open the Simulation Data Inspector.
7. In the Simulation Data Inspector, rename the run to indicate that it is for the unprotected model.
8. In the Simulink Editor, right-click the `CounterA` block. From the context menu, select `Block Parameters (ModelReference)`. In the Block Parameters dialog

box, specify the **Model name** parameter with the name of the protected model, `sldemo_mdlref_counter.slxp`. A shield icon appears on the Model block. Repeat this for `CounterB` block and `CounterC` block.

9. Simulate the model, which now refers to the protected model. When the simulation is complete, a new run appears in the Simulation Data Inspector.
10. In the Simulation Data Inspector, rename the new run to indicate that it is for the protected model.
11. In the Simulation Data Inspector, click the **Compare** tab. From the **Baseline** and **Compare To** lists, select the runs from the unprotected and protected model, respectively. Click **Compare Runs** to compare the runs. For more information about comparing runs, see "Compare Simulation Data" (Simulink).

Save Base Workspace Definitions

Referenced models might use object definitions or tunable parameters that are defined in the MATLAB base workspace. These variables are not saved with the model. When you protect a model, you must obtain the definitions of required base workspace entities and ship them with the model.

The following base workspace variables must be saved to a MAT-file:

- Global tunable parameter
- Global data store
- The following objects used by a signal that connects to a root-level model Inport or Outport:
  - `Simulink.Signal`
  - `Simulink.Bus`
  - `Simulink.Alias`
  - `Simulink.NumericType` that is an alias

For more information, see "Edit and Manage Workspace Variables Used by Models" (Simulink).

Before executing the protected model as a part of a third-party model, the receiver of the protected model must load the MAT-file.

Package a Protected Model

In addition to the protected model file (`.slxp`), you might need to include additional files in the protected model package:

- Harness model file.
- Any required definitions saved in a MAT-file. For more information, see “Save Base Workspace Definitions” on page 8-22.
- Instructions on how to retrieve the files.

Some ways to deliver the protected model package are:

- Provide the `.slxp` file and other supporting files as separate files.
- Combine the files into a ZIP or other container file.
- Combine the files using a manifest. For more information, see “Export Files in a Manifest” (Simulink).
- Provide the files in some other standard or proprietary format specified by the receiver.

Whichever approach you use to deliver a protected model, include information on how to retrieve the original files. One approach to consider is to use the Simulink Manifest Tools, as described in “Analyze Model Dependencies” (Simulink).

Specify Custom Obfuscator for Protected Model

When creating a protected model, you can specify your own postprocessing function for files that the protected model creation process generates. Prior to packaging the protected model files, this function is called by the `Simulink.ModelReference.protect` function. You can use this functionality to run your own custom obfuscator on the generated files by following these steps:

1. Create your postprocessing function. Use this function to call your custom obfuscator. The function must be on the MATLAB path and accept a Simulink.ModelReference.ProtectedModel.HookInfo object as an input variable.
2. In your function, get the files and exported symbol information that your custom obfuscator requires to process the protected model files. To get the files and information, access the properties of your function input variable. The variable is a Simulink.ModelReference.ProtectedModel.HookInfo object with the following properties:

   - `SourceFiles`
   - `NonSourceFiles`
   - `ExportedSymbols`
3. Pass the protected model file information to your custom obfuscator. The following is an example of a postprocessing function for custom obfuscation:

```
function myHook(protectedModelInfo)

    % Get source file list information.
    srcFileList = protectedModelInfo.SourceFiles;
    disp('### Obfuscating...');
    for i=1:length(srcFileList)
        disp(['### Obfuscator: Processing ' srcFileList{i} '...']);
        % call to custom obfuscator
        customObfuscator(srcFileList{i});
    end
end
```

4. Specify your postprocessing function when creating the protected model:

```
Simulink.ModelReference.protect('myModel, 'Mode', 'CodeGeneration', ...
    'CustomPostProcessingHook', ...
    @(protectedModelInfo)myHook(protectedModelInfo))
```

The creator of the protected model also has the option of enabling obfuscation of simulation target code and generated code through the '`ObfuscateCode`' option of the `Simulink.ModelReference.protect` function. Your custom obfuscator runs only on the generated code and not on the simulation target code. If both obfuscators are in use, the custom obfuscator is the last to run on the generated code before the files are packaged.

Define Callbacks for Protected Model

When you create a protected model, you can customize its behavior by defining callbacks. Callbacks specify code that executes when a protected model user views, simulates, or generates code for the protected model. A protected model user cannot view or modify a callback. To create a protected model with callbacks:

1. Define Simulink.ProtectedModel.Callback objects for each callback.
2. To create your protected model, call the `Simulink.ModelReference.protect` function. Use the `'Callbacks'` option to specify a cell array of callbacks to include in the protected model.

In this section...

Creating Callbacks

To create and define a protected model callback, create a Simulink.ProtectedModel.Callback object. Callback objects specify:

- The code to execute for the callback. The code can be a character vector of MATLAB commands or a script on the MATLAB path.
- The event that triggers the callback. The event can be `'PreAccess'` or `'Build'`.
- The protected model functionality that the event applies to. The functionality can be `'CODEGEN'`, `'SIM'`, `'VIEW'`, or `'AUTO'`. If you select `'AUTO'`, and the event is `'PreAccess'`, the callback applies to each functionality. If you select `'AUTO'`, and the event is `'Build'`, the callback applies only to `'CODEGEN'` functionality. If you do not select any functionality, the default behavior is `'AUTO'`.
- The option to override the protected model build process. This option applies only to `'CODEGEN'` functionality.

You can create only one callback per event and per functionality.

Defining Callback Code

You can define the code for a callback by using either a character vector of MATLAB commands or a script on the MATLAB path. When you write callback code, follow these guidelines:

- Callbacks must use MATLAB code (`.m` or `.p`).
- The code can include protected model functions or any MATLAB command that does not require loading the model.
- Callback code must not call out to external utilities unless those utilities are available in the environment where the protected model is used.
- Callback code cannot reference the source protected model unless you are using protected model functions.

You can use the `Simulink.ProtectedModel.getCallbackInfo` function in callback code to get information on the protected model. The function returns a Simulink.ProtectedModel.CallbackInfo object that provides the protected model name and the names of submodels. If the callback is specified for `'CODEGEN'` functionality and `'Build'` event, the object provides the target identifier and model code interface type (`'Top model'` or `'Model reference'`).

Create a Protected Model with Callbacks

This example creates a protected model with a callback for code generation.

1. On the MATLAB path, create a callback script, `pm_callback.m`, containing:

   ```
   s1 = 'Code interface is: ';
   cbinfobj = Simulink.ProtectedModel.getCallbackInfo(...
   'sldemo_mdlref_counter','Build','CODEGEN');
   disp([s1 cbinfobj.CodeInterface]);
   ```

2. Create a callback that uses the script. If the callback code replaces the protected model build process, set the override option.

   ```
   pmCallback = Simulink.ProtectedModel.Callback('Build',...
   'CODEGEN', 'pm_callback.m');
   pmCallback.setOverrideBuild(true);
   ```

3. Create the protected model and specify the code generation callback.

   ```
   Simulink.ModelReference.protect('sldemo_mdlref_counter',...
   ```

```
'Mode', 'CodeGeneration','Callbacks',{pmCallback})
```

4 Build the protected model. In place of the build, the callback displays the code interface.

```
rtwbuild('sldemo_mdlref_basic')
```

See Also

Simulink.ProtectedModel.Callback | `Simulink.ModelReference.protect` | `Simulink.ProtectedModel.getCallbackInfo`

More About

- "Protect a Referenced Model" (Simulink Coder)
- "Code Generation Support in a Protected Model" (Simulink Coder)

9

Component Initialization, Reset, and Termination in Simulink Coder

Generate Code That Responds to Initialize, Reset, and Terminate Events

To generate code from a modeling component that responds to initialize, reset, and terminate events during execution, use the blocks Initialize Function and Terminate Function. For information on how to use these blocks, see "Create Model to Initialize, Reset, and Terminate State" (Simulink). You can use the blocks anywhere in a model hierarchy.

Examples of when to generate code that responds to initialize, reset, or terminate events include:

- Starting and stopping a component.
- Calculating initial conditions.
- Saving and restoring state from nonvolatile memory.
- Generating reset entry-point functions that respond to external events.

Each nonvirtual subsystem and referenced model can have its own set of initialize, reset, and terminate functions.

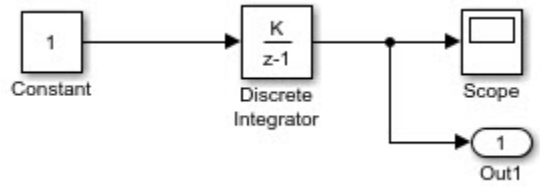
The code generator produces initialization and termination code differently than reset code. For initialization and termination code, the code generator includes your component initialization and termination code in the default entry-point functions, *`model`*`_initialize` and *`model`*`_terminate`. The code generator produces reset code only if you model reset behavior.

Generate Code for Initialize and Terminate Events

When you generate code for a component that includes Initialize Function and Terminate Function blocks, the code generator:

- Includes initialize event code with default initialize code in entry-point function *`model`*`_initialize`.
- Includes terminate event code with default terminate code in entry-point function *`model`*`_terminate`.

Consider the model rtwdemo_irt_base.



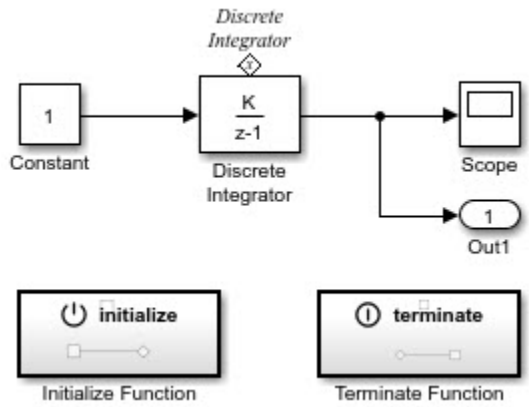
For this model, the code generator produces initialize and terminate entry-point functions that other code can interface with.

```
void rtwdemo_irt_base_initialize(void)
void rtwdemo_irt_base_terminate(void)
```

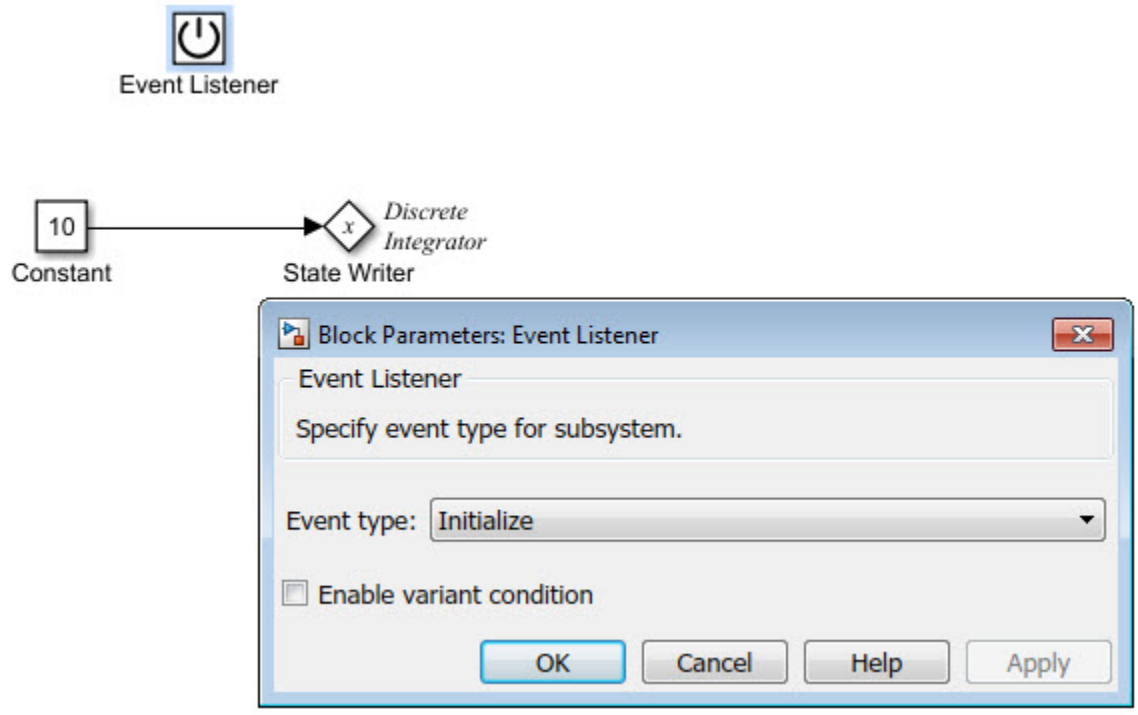
This code appears in the generated file `rtwdemo_irt_base.c`. The initialize function, `rtwdemo_irt_base_initialize`, initializes an error status. The terminate function, `rtwdemo_irt_base_terminate`, requires no code. This code assumes that support for nonfinite numbers and MAT-file logging is disabled.

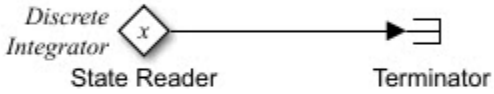
```
void rtwdemo_irt_base_initialize(void)
{
   rtmSetErrorStatus(rtwdemo_irt_base_M, (NULL));

   (void) memset((void *)&rtwdemo_irt_base_DW, 0,
                 sizeof(DW_rtwdemo_irt_base_T));

   rtwdemo_irt_base_Y.Out1 = 0.0;

   rtwdemo_irt_base_DW.DiscreteIntegrator_DSTATE = 0.0;
}

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

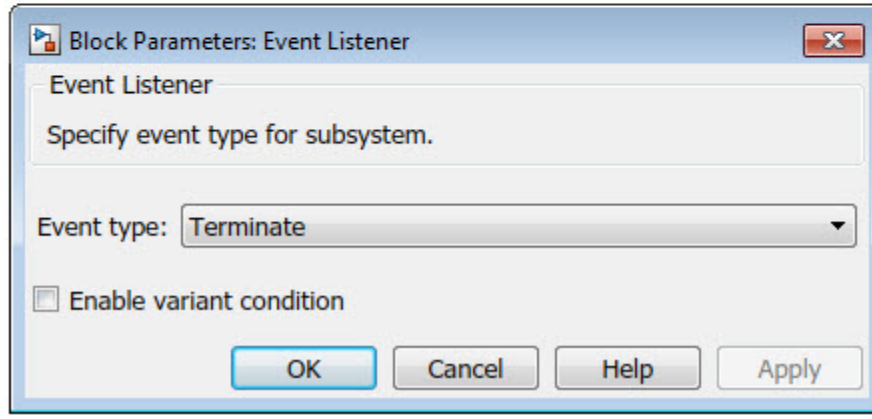
Add Initialize Function and Terminate Function blocks to the model (see `rtwdemo_irt_initterm`). The Initialize Function block uses the State Writer block to set the initial condition of a Discrete Integrator block. The Terminate Function block includes a State Reader block, which reads the state of the Discrete Integrator block.



The **Event type** parameter of the Event Listener block for the initialize and terminate functions is set to `Initialize` and `Terminate`, respectively. The initialize function uses the State Writer block to initialize the input value for the Discrete Integrator block to 10. The terminate function uses the State Reader block to read the state of the Discrete Integrator block.


Event Listener
10
Constant
x
Discrete
Integrator
State Writer
Block Parameters: Event Listener
Event Listener
Specify event type for subsystem.
Event type:
Initialize
Enable variant condition
OK
Cancel
Help
Apply





The code generator includes the event code that it produces for the Initialize Function and Terminate Function blocks with standard initialize and terminate code in entry-point functions `rtwdemo_irt_initterm_initialize` and `rtwdemo_irt_initterm_terminate`. This code assumes that support for nonfinite numbers and MAT-file logging is disabled.

```
void rtwdemo_irt_initterm_initialize(void)
{
   rtmSetErrorStatus(rtwdemo_irt_initterm_M, (NULL));

   (void) memset((void *)&rtwdemo_irt_initterm_DW, 0,
                 sizeof(DW_rtwdemo_irt_initterm_T));

   rtwdemo_irt_initterm_Y.Out1 = 0.0;
```

```
    rtwdemo_irt_initterm_DW.DiscreteIntegrator_DSTATE = 10.0;
}

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

The initialization code:

- Initializes an error status.
- Allocates memory for block I/O and state parameters.
- Assigns the value of the constant input parameter to the state parameter of the discrete integrator.

The termination code assigns the value of the discrete integrator state parameter to the block I/O parameter.

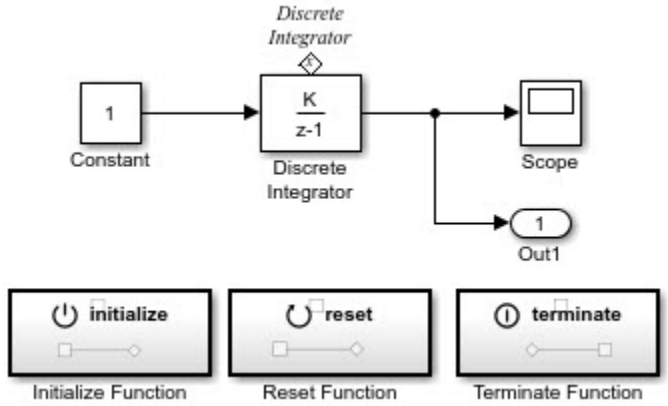
Generate Code for Reset Events

Generate code that responds to a reset event by including an Initialize Function or Terminate Function block in a modeling component. Configure the block for a reset by setting the **Event type** parameter of its Event Listener block to `Reset`. Also set the **Event name** parameter. The default name is `reset`.

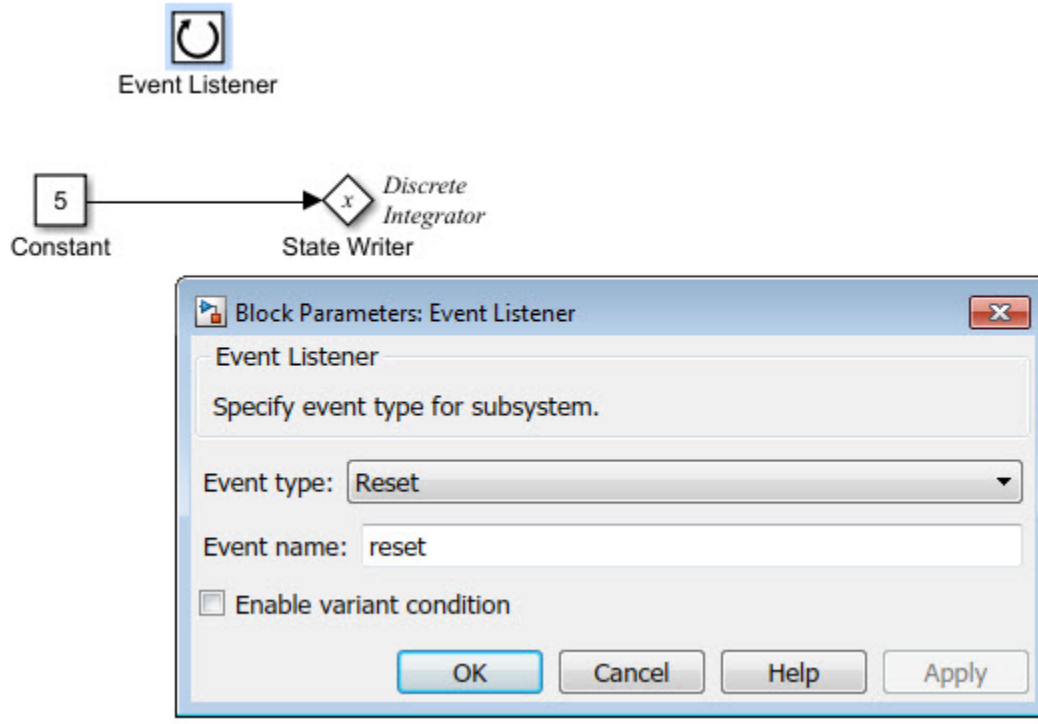
The code generator produces a reset entry-point function *only* if you model reset behavior. If a component contains multiple reset specifications, the code that the code generator produces depends on whether reset functions share an event name. For a given component hierarchy:

- For reset functions with unique event names, the code generator produces a separate entry-point function for each named event. The name of each function is the name of the corresponding event.
- For reset functions that share an event name, the code generator aggregates the reset code into one entry-point function. The code for the reset functions appears in order, starting with the lowest level (innermost) of the component hierarchy and ending with the root (outermost). The name of the function is *`model`*`_reset`. For more information, see “Event Names and Code Aggregation” (Simulink Coder).

Consider the model `rtwdemo_irt_reset`, which includes a Reset Function block derived from an Initialize Function block.



The **Event type** and **Event name** parameters of the Event Listener block are set to `Reset` and `reset`, respectively. The function uses the State Writer block to reset the input value for the Discrete Integrator block to 5.



The code generator produces reset function `rtwdemo_irt_reset_reset`. The code for the function appears in the `Real-time model` section of the *`model`*`.c` file.

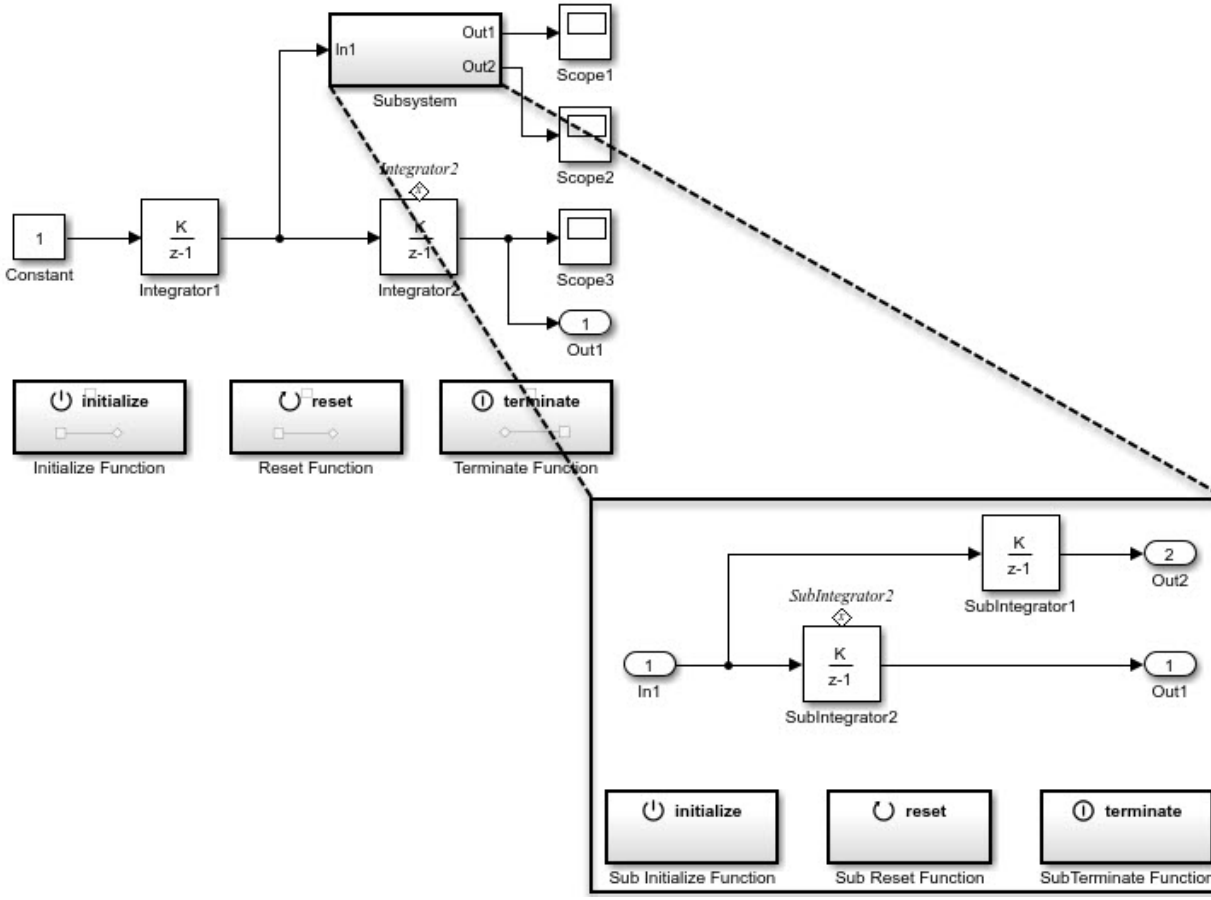
```
void rtwdemo_irt_reset_reset(void)

{
  rtwdemo_irt_reset_DW.DiscreteIntegrator_DSTATE = 5.0;
}
```

Event Names and Code Aggregation

Use the Initialize Function and Terminate Function blocks to define multiple initialize, reset, and terminate functions for a component hierarchy. Define only one initialize function and one terminate function per hierarchy level. You can define multiple reset functions for a hierarchy level. The event names that you configure for the functions at a given level must be unique.

When producing code, the code generator aggregates code for functions that have a given event name across the entire component hierarchy into one entry-point function. The code for reset functions appears in order, starting with the lowest level (innermost) of the component hierarchy and ending with the root (outermost). The code generator uses the event name to name the function.

For example, the model `rtwdemo_irt_shared` includes a subsystem that replicates the initialize, reset, and terminate functions that are in the parent model.



Although the model includes multiple copies of the initialize, reset, and terminate functions, the code generator produces one entry-

point function for reset (`rtwdemo_irt_shared_reset`), one for initialize (`rtwdemo_irt_shared_initialize`), and one for terminate (`rtwdemo_irt_shared_terminate`). Within each entry-point function, after listing code for blocks configured with an initial condition (*`model`*`_P.`*`block`*`_IC`), the code generator orders code for components, starting with the lowest level of the hierarchy and ending with the root.

```
.
.
.
void rtwdemo_irt_shared_reset(void)
{
   rtwdemo_irt_shared_DW.SubIntegrator2_DSTATE =
     rtwdemo_irt_shared_P.Constant1_Value;

   rtwdemo_irt_shared_DW.SubIntegrator2_DSTATE = 5.0;

   rtwdemo_irt_shared_DW.Integrator2_DSTATE = 5.0;
}
.
.
.
void rtwdemo_irt_shared_initialize(void)
{
  rtmSetErrorStatus(rtwdemo_irt_shared_M, (NULL));

  (void) memset(((void *)&rtwdemo_irt_shared_DW), 0,
                sizeof(DW_rtwdemo_irt_shared_T));

  rtwdemo_irt_shared_Y.Out1 = 0.0;

  rtwdemo_irt_shared_DW.Integrator1_DSTATE = 0.0;

  rtwdemo_irt_shared_DW.SubIntegrator2_DSTATE = 2.0;

  rtwdemo_irt_shared_DW.Integrator2_DSTATE = 10.0;
.
.
.
void rtwdemo_irt_shared_terminate(void)
{
   /* (no terminate code required) */
```

```
   }
```

If you rename the event configured for the subsystem reset function to `reset_02`, the code generator produces two reset entry-point functions, `rtwdemo_irt_shared_reset` and `rtwdemo_irt_shared_reset_02`.

```
void rtwdemo_irt_shared_reset(void)
{
   rtwdemo_irt_shared_DW.Integrator2_DSTATE = 5.0;
}

void rtwdemo_irt_shared_reset_02(void)
{
   rtwdemo_irt_shared_DW.SubIntegrator2_DSTATE = 5.0;
}
```

Limitations

You cannot generate code from a:

- Harness model—a root model that contains a Model block, which exposes initialize, reset, or terminate function ports.
- Model configured for C++ code generation.

Related Examples

- "Create Model to Initialize, Reset, and Terminate State" (Simulink)
- "Entry-Point Functions and Scheduling" (Simulink Coder)
- "Initialization of Signal, State, and Parameter Data in the Generated Code" on page 19-165

Stateflow Blocks in Simulink Coder

Code Generation of Stateflow Blocks

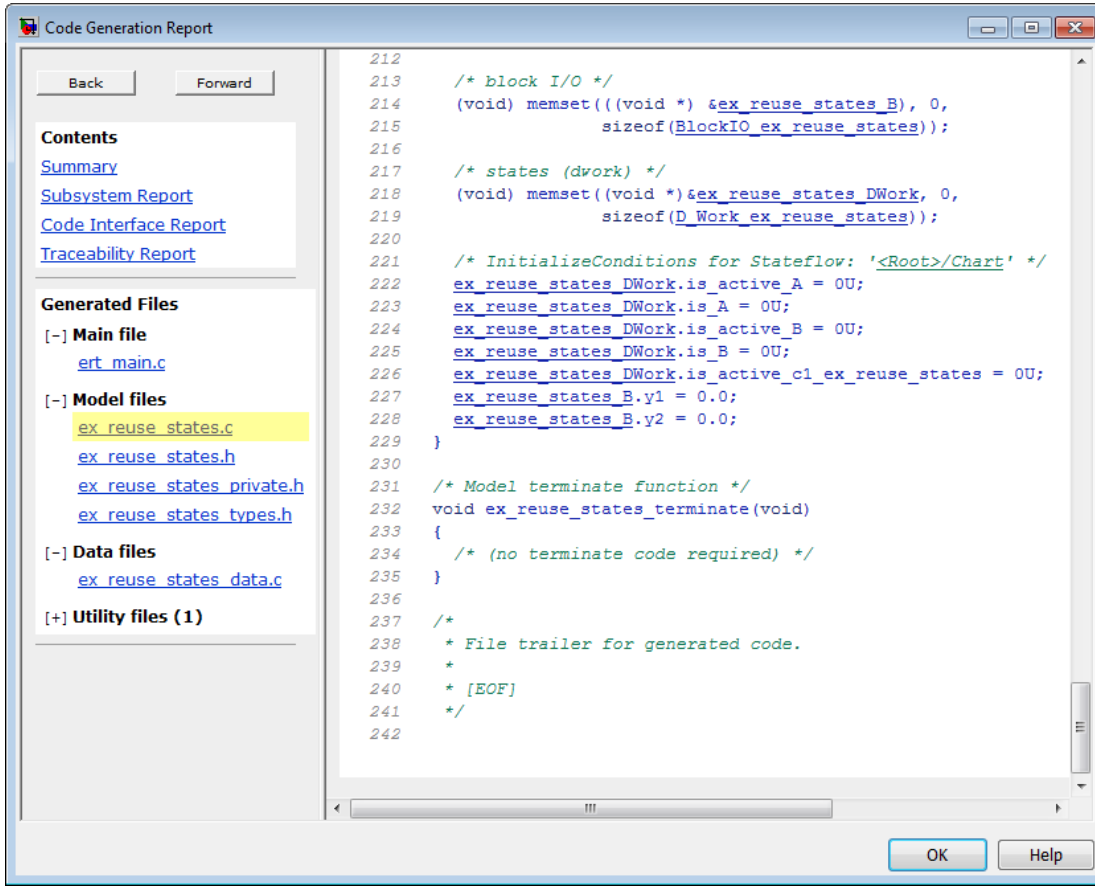
The code generator produces code for Stateflow blocks for rapid prototyping. If you have an Embedded Coder license, you can generate production code for Stateflow blocks.

Comparison of Code Generation Methods

The following sections compare two ways of generating code.

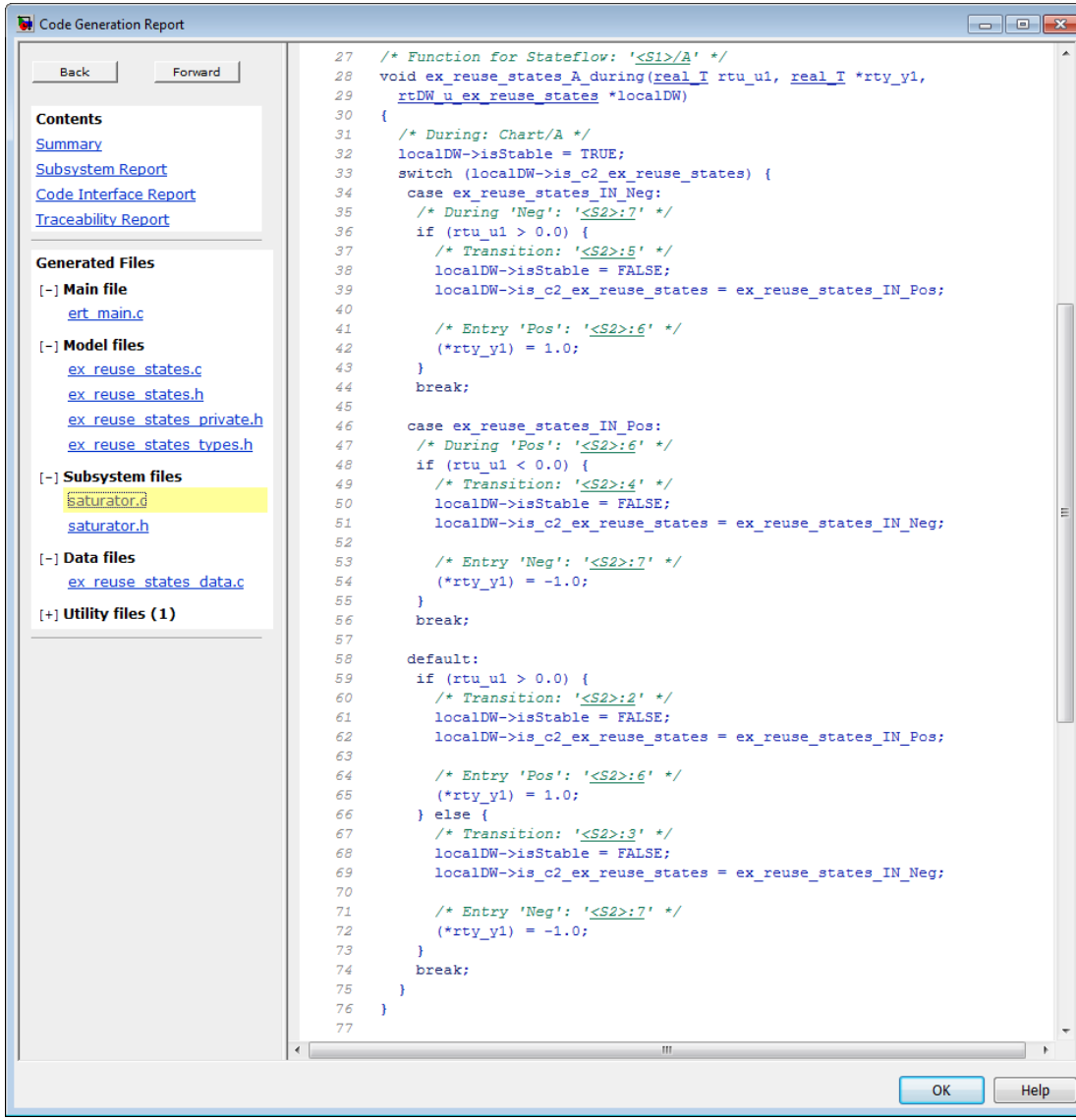
Code Generation Without Atomic Subcharts

You generate code for the entire model in one file and look through that entire file to find code for a specific part of the chart.



Code Generation With Atomic Subcharts

You specify code generation parameters so that code for an atomic subchart appears in a separate file. This method of code generation enables unit testing for a specific part of a chart. You can avoid searching through unrelated code and focus only on the part that interests you.



Note: Unreachable Stateflow states are optimized out and are not included in the generated code.

For more information, see “Generate Reusable Code for Unit Testing” on page 10-8.

Generate Reusable Code for Atomic Subcharts

In this section...

How to Generate Reusable Code for Linked Atomic Subcharts

To specify code generation parameters for linked atomic subcharts from the same library:

1. Open the library model that contains your atomic subchart.
2. Unlock the library.
3. Right-click the library chart and select **Block Parameters**.
4. In the dialog box, specify the following parameters:
   a. On the **Main** tab, select **Treat as atomic unit**.
   b. On the **Code Generation** tab, set **Function packaging** to `Reusable function`.
   c. Set **File name options** to `User specified`.
   d. For **File name**, enter the name of the file with no extension.
   e. Click **OK** to apply the changes.
5. (OPTIONAL) Customize the generated function names for atomic subcharts:
   a. Open the Model Configuration Parameters dialog box.
   b. On the **Code Generation** pane, set **System target file** to `ert.tlc`.
   c. Navigate to the **Code Generation > Symbols** pane.
   d. For **Subsystem methods**, specify the format of the function names using a combination of the following tokens:
      - `$R` — root model name
      - `$F` — type of interface function for the atomic subchart
      - `$N` — block name
      - `$H` — subsystem index
      - `$M` — name-mangling text

e Click **OK** to apply the changes.

When you generate code for your model, a separate file stores the code for linked atomic subcharts from the same library.

How to Generate Reusable Code for Unlinked Atomic Subcharts

To specify code generation parameters for an unlinked atomic subchart:

1 In your chart, right-click the atomic subchart and select **Properties**.

2 In the dialog box, specify the following parameters:

   a Set **Code generation function packaging** to `Reusable function`.

   b Set **Code generation file name options** to `User specified`.

   c For **Code generation file name**, enter the name of the file with no extension.

   d Click **OK** to apply the changes.

3 (OPTIONAL) Customize the generated function names for atomic subcharts:

   a Open the Model Configuration Parameters dialog box.

   b On the **Code Generation** pane, set **System target file** to `ert.tlc`.

   c Navigate to the **Code Generation > Symbols** pane.

   d For **Subsystem methods**, specify the format of the function names using a combination of the following tokens:

      - `$R` — root model name
      - `$F` — type of interface function for the atomic subchart
      - `$N` — block name
      - `$H` — subsystem index
      - `$M` — name-mangling text

   e Click **OK** to apply the changes.

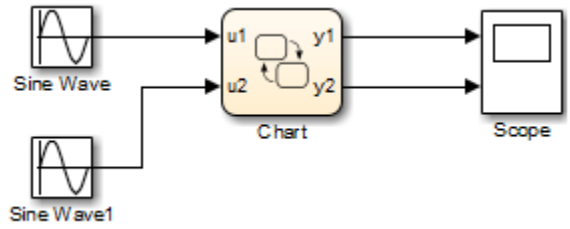
When you generate code for your model, a separate file stores the code for the atomic subchart. For more information, see “Generate Reusable Code for Unit Testing” on page 10-8.

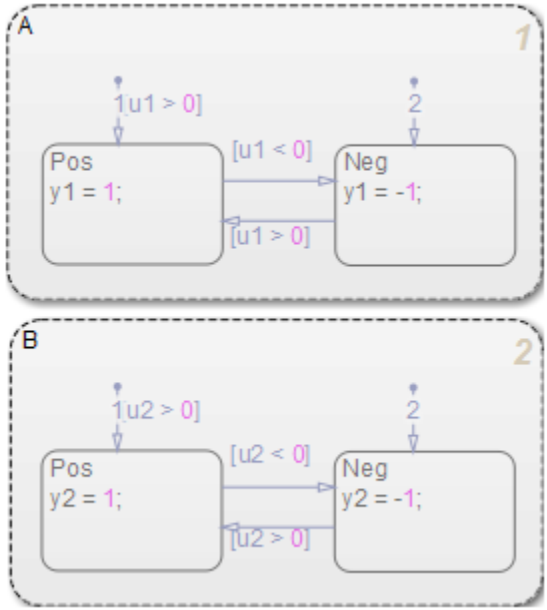
Generate Reusable Code for Unit Testing

In this section...

Goal of the Tutorial

Assume that you have the following model, and the chart has two states:

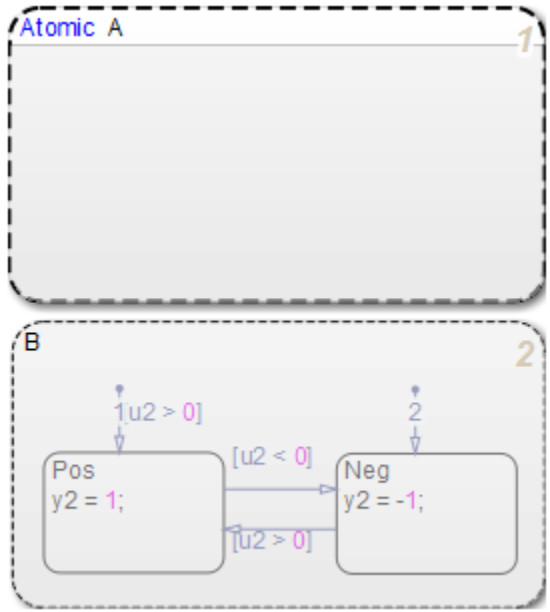




Suppose that you want to generate reusable code so that you can perform unit testing on state A. You can convert that part of the chart to an atomic subchart and then specify a separate file to store the generated code.

Convert a State to an Atomic Subchart

To convert state A to an atomic subchart, right-click the state and select **Group & Subchart > Atomic Subchart**. State A changes to an atomic subchart:



Specify Code Generation Parameters

Set Up a Standalone C File for the Atomic Subchart

1. Open the properties dialog box for A.
2. Set **Code generation function packaging** to `Reusable function`.
3. Set **Code generation file name options** to `User specified`.
4. For **Code generation file name**, enter `saturator` as the name of the file.
5. Click **OK**.

Set Up the Code Generation Report

1. Open the Model Configuration Parameters dialog box.
2. In the **Code Generation** pane, set **System target file** to `ert.tlc`.
3. In the **Code Generation** > **Report** pane, select **Create code generation report**.

   This step automatically selects **Open report automatically** and **Code-to-model** on the **All Parameters** tab.

4 Select **Model-to-code** on the **All Parameters** tab.

5 Click **Apply**.

Customize the Generated Function Names

1 In the Model Configuration Parameters dialog box, go to the **Code Generation** > **Symbols** pane.

2 Set **Subsystem methods** to the format scheme `$R$N$M$F`, where:

- `$R` is the root model name.
- `$N` is the block name.
- `$M` is the mangle token.
- `$F` is the type of interface function for the atomic subchart.

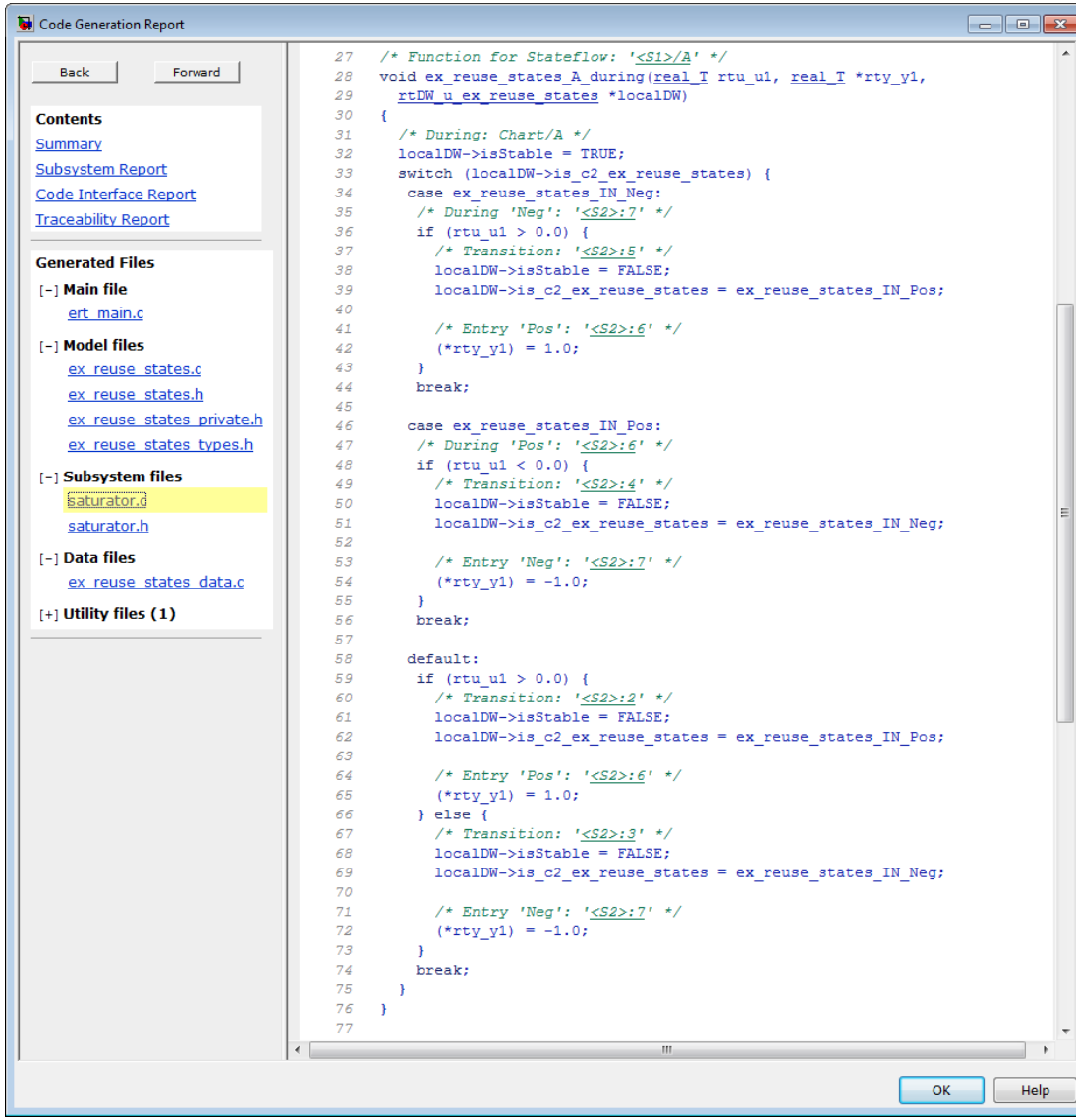
For more information, see "Subsystem methods" (Simulink Coder).

3 Click **Apply**.

Generate Code for Only the Atomic Subchart

To generate code for your model, press **Ctrl+B**. In the code generation report that appears, you see a separate file that contains the generated code for the atomic subchart.

To inspect the code for `saturator.c`, click the hyperlink in the report to see the following code:



Line 28 shows that the `during` function generated for the atomic subchart has the name `ex_reuse_states_A_during`. This name follows the format scheme `$R$N$M$F` specified for **Subsystem methods**:

- `$R` is the root model name, `ex_reuse_states`.
- `$N` is the block name, `A`.
- `$M` is the mangle token, which is empty.
- `$F` is the type of interface function for the atomic subchart, `during`.

Note: The line numbers shown can differ from the numbers that appear in your code generation report.

Inline State Functions in Generated Code

In this section...

Inlined Generated Code for State Functions

By default, the code generator uses an internal heuristic to determine whether to inline generated code for state functions. The heuristic takes into consideration an inlining threshold. As code grows and shrinks in size, generated code for state functions can be unpredictable.

If your model includes Stateflow objects and you have rigorous requirements for traceability between generated code and the corresponding state functions, you can override the default behavior. Use the state property `Function Inline Option` to explicitly force or prevent inlining of state functions.

What Happens When You Force Inlining

If you force inlining for a state, the code generator inlines code for state actions into the parent function. The parent function contains code for executing the state actions, outer transitions, and flow charts. It does not include code for empty state actions.

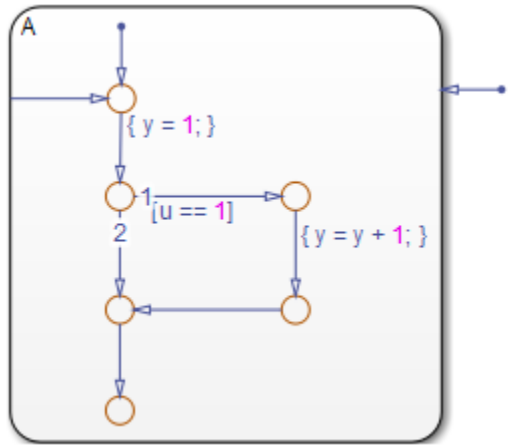
What Happens When You Prevent Inlining

If you prevent inlining for a state, the code generator produces these static functions for state *foo*.

Function	Description
`enter_atomic_`*`foo`*	Marks *`foo`* active and performs entry actions.
`enter_internal_`*`foo`*	Calls default paths.
`inner_default_`*`foo`*	Executes flow charts that originate when an inner transition and default transition reach the same junction inside a state.

Function	Description
	The code generator produces this function only when the flow chart is complex enough to exceed the inlining threshold. In generated code, Stateflow software calls this function from both the `enter_internal_foo` and `foo` functions.
`foo`	Checks for valid outer transitions and if none, performs during actions.
`exit_atomic_foo`	Performs exit actions and marks `foo` inactive.
`exit_internal_foo`	Performs exit actions of the child substates and then exits `foo`.

Suppose the following chart is in model M.



If you prevent inlining for state A, the code generator produces this code.

```
static void M_inner_default_A(void);
static void M_exit_atomic_A(void);
static void M_A(void);
static void M_enter_atomic_A(void);
static void M_enter_internal_A(void);
```

How to Set the State Function Inline Option

To set the function inlining property for a state:

1 Right-click inside the state and, from the context menu, select **Properties**.

The State properties dialog box opens.

2 In the **Function Inline Option** field, select one of these options.

Option	Behavior
Inline	Forces inlining of state functions into the parent function, as long as the function is not part of a recursion. See "What Happens When You Force Inlining" on page 10-14.
Function	Prevents inlining of state functions. Generates up to six static functions for the state. See "What Happens When You Prevent Inlining" on page 10-14.
Auto	Uses internal heuristics to determine whether or not to inline the state functions.

3 Click **Apply**.

Best Practices for Controlling State Function Inlining

To	Set Function Inline Option Property To
Generate a separate function for each action of a state and a separate function for each action of its substates	**Function** for the state and each substate
Generate a separate function for each action of a state, but include code for the associated action of its substates	**Function** for the state and **Inline** for each substate

Air-Fuel Ratio Control System with Stateflow Charts

Generate code for an air-fuel ratio control system designed with Simulink® and Stateflow®.

Figures 1, 2, and 3 show relevant portions of the sldemo_fuelsys model, a closed-loop system containing a plant and controller. The plant validates the controller in simulation early in the design cycle. In this example, you generate code for the relevant controller subsystem, "fuel_rate_control". Figure 1 shows the top-level simulation model.

Open sldemo_fuelsys via rtwdemo_fuelsys and compile the diagram to see the signal data types.

```
rtwdemo_fuelsys
sldemo_fuelsys([],[],[],'compile');
sldemo_fuelsys([],[],[],'term');
```

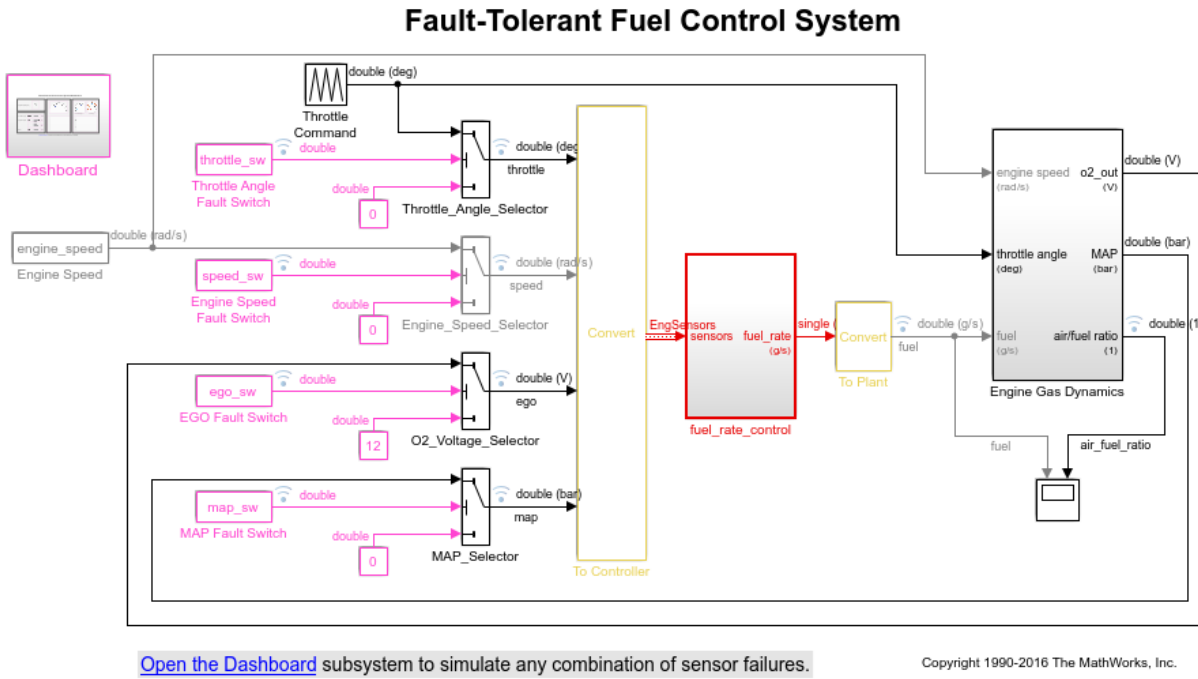


Figure 1: Top-level model of the plant and controller

The air-fuel ratio control system is comprised of Simulink® and Stateflow®. The control system is the portion of the model for which you generate code.

```
open_system('sldemo_fuelsys/fuel_rate_control');
```

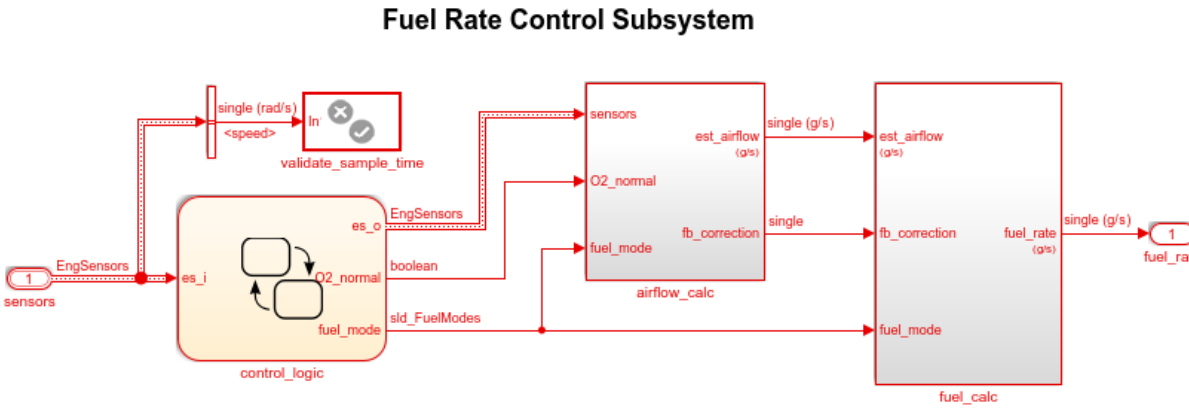


Figure 2: The air-fuel ratio controller subsystem

The control logic is a Stateflow® chart that specifies the different modes of operation.

```
open_system('sldemo_fuelsys/fuel_rate_control/control_logic');
```

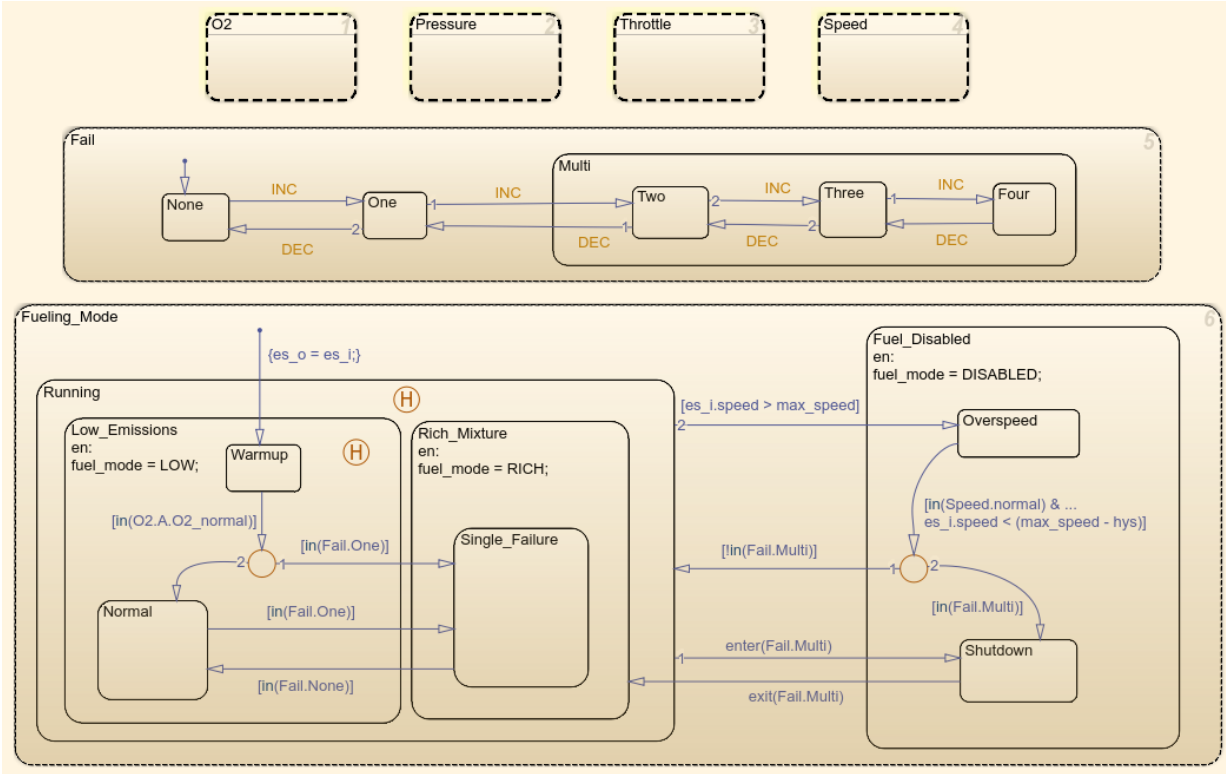


Figure 3: Air-fuel rate controller logic

Close these windows.

```
close_system('sldemo_fuelsys/fuel_rate_control/airflow_calc');
close_system('sldemo_fuelsys/fuel_rate_control/fuel_calc');
close_system('sldemo_fuelsys/fuel_rate_control/control_logic');
hDemo.rt=sfroot;hDemo.m=hDemo.rt.find('-isa','Simulink.BlockDiagram');
hDemo.c=hDemo.m.find('-isa','Stateflow.Chart','-and','Name','control_logic');
hDemo.c.visible=false;
close_system('sldemo_fuelsys/fuel_rate_control');
```

Configure and Build the Model with Simulink® Coder™

Simulink® Coder™ generates generic ANSI® C code for Simulink® and Stateflow® models via the Generic Real-Time (GRT) target. You can configure a model for code generation programmatically.

```
rtwconfiguredemo('sldemo_fuelsys','GRT');
```

For this example, build only the air-fuel ratio control system. Once the code generation process is complete, an HTML report detailing the generated code is displayed. The main body of the code is located in fuel_rate_control.c.

```
rtwbuild('sldemo_fuelsys/fuel_rate_control');
```

```
### Starting build procedure for model: fuel_rate_control
### Successful completion of build procedure for model: fuel_rate_control
```

Configure and Build the Model with Embedded Coder®

Embedded Coder® generates production ANSI® C/C++ code via the Embedded Real-Time (ERT) target. You can configure a model for code generation programmatically.

```
rtwconfiguredemo('sldemo_fuelsys','ERT');
```

Repeat the build process and inspect the generated code. In the Simulink® Coder™ Report, you can navigate to the relevant code segments interactively by using the **Previous** and **Next** buttons. From the chart context menu (right-click the Stateflow® block), select **Code Generation > Navigate to Code**. Programmatically, use the rtwtrace utility.

```
rtwbuild('sldemo_fuelsys/fuel_rate_control');
rtwtrace('sldemo_fuelsys/fuel_rate_control/control_logic')
```

```
### Starting build procedure for model: fuel_rate_control
### Successful completion of build procedure for model: fuel_rate_control
```

View the air-fuel ratio control logic in the generated code.

```
rtwdemodbtype('fuel_rate_control_ert_rtw/fuel_rate_control.c','/* Function for Chart:'
```

```
/* Function for Chart: '<S1>/control_logic' */
static void Fueling_Mode(const int32_T *sfEvent)
```

```
{
  /* During 'Fueling_Mode': '<S3>:21' */
  /* This state interprets the other states in the chart to directly control the fueli
  switch (rtDW.bitsForTID0.is_Fueling_Mode) {
   case IN_Fuel_Disabled:
    rtDW.fuel_mode = DISABLED;

    /* During 'Fuel_Disabled': '<S3>:22' */
    /* The fuel is completely shut off while in this state. */
    switch (rtDW.bitsForTID0.is_Fuel_Disabled) {
     case IN_Overspeed:
      /* Inport: '<Root>/sensors' */
      /* During 'Overspeed': '<S3>:24' */
      /* The speed is dangerously high, so shut off the fuel. */
      if ((rtDW.bitsForTID0.is_Speed == IN_normal) && (rtU.sensors.speed <
           603.0F)) {
        /* Transition: '<S3>:54' */
        if (!(rtDW.bitsForTID0.is_Fail == IN_Multi)) {
          /* Transition: '<S3>:55' */
          rtDW.bitsForTID0.is_Fuel_Disabled = IN_NO_ACTIVE_CHILD;
          rtDW.bitsForTID0.is_Fueling_Mode = IN_Running;

          /* Entry Internal 'Running': '<S3>:23' */
          switch (rtDW.bitsForTID0.was_Running) {
           case IN_Low_Emissions:
            if (rtDW.bitsForTID0.is_Running != IN_Low_Emissions) {
              rtDW.bitsForTID0.is_Running = IN_Low_Emissions;
              rtDW.bitsForTID0.was_Running = IN_Low_Emissions;

              /* Entry 'Low_Emissions': '<S3>:25' */
              rtDW.fuel_mode = LOW;
            }

            /* Entry Internal 'Low_Emissions': '<S3>:25' */
            switch (rtDW.bitsForTID0.was_Low_Emissions) {
             case IN_Normal:
              rtDW.bitsForTID0.is_Low_Emissions = IN_Normal;
              rtDW.bitsForTID0.was_Low_Emissions = IN_Normal;
              break;
```

Close the model and code generation report.

```
clear hDemo;
rtwdemoclean;
close_system('sldemo_fuelsys',0);
```

Related Examples

For related fixed-point examples that use sldemo_fuelsys, see

- **Fixed-point design** - “Fixed-Point Fuel Rate Control System” (Fixed-Point Designer)
- **Fixed-point production C/C++ code generation** - “Air-Fuel Ratio Control System with Fixed-Point Data” (Simulink Coder)

11

Block Authoring and Code Generation for Simulink Coder

S-Functions and Code Generation

In this section...

You use S-functions to extend Simulink support for simulation and code generation. For example, you can use them to:

- Represent custom algorithms
- Interface existing external code with Simulink and the code generator
- Represent device drivers for interfacing with hardware
- Generate highly optimized code for embedded systems
- Verify code generated for a subsystem as part of a Simulink simulation

The application program interface (API) for writing S-functions allows you to implement generic algorithms in the Simulink environment with a great deal of flexibility. If you intend to use S-functions in a model for code generation, the level of flexibility can vary. For example, it is not possible to access the MATLAB workspace from an S-function that you use with the code generator. This topic explains conditions to be aware of for using S-functions However, using the techniques presented in this topic, you can create S-functions for most applications that work with the generated code.

Although S-functions provide a generic and flexible solution for implementing complex algorithms in a model, the underlying API incurs overhead in terms of memory and computation resources. Most often the additional resources are acceptable for real-time rapid prototyping systems. In many cases, though, additional resources are unavailable in real-time embedded applications. You can minimize memory and computational requirements by using the Target Language Compiler technology provided with the code generator to inline your S-functions. If you are producing an S-function for existing external code, consider using the Legacy Code Tool to generate your S-function and relevant TLC file.

This content assumes that you understand the following concepts:

- Level 2 S-functions

- Target Language Compiler (TLC) scripting
- How the code generator produces and builds C/C++ code

Notes This information is for code generator users. Even if you do not currently use the code generator, follow these practices when writing S-functions, especially if you are creating general-purpose S-functions.

Types of S-Functions

Examples for which you might choose to implement an S-function for simulation and code generation include:

1. "I'm not concerned with efficiency. I just want to write one version of my algorithm and have it work in the Simulink and code generator products automatically."
2. "I want to implement a highly optimized algorithm in the Simulink and code generator products that looks like a built-in block and generates very efficient code."
3. "I have a lot of hand-written code that I need to interface. I want to call my function from the Simulink and code generator products in an efficient manner."

Respectively, the preceding situations map to the following MathWorks terminology:

1. Noninlined S-function
2. Inlined S-function
3. Autogenerated S-function for external code

Noninlined S-Functions

A noninlined S-function is a C or C++ MEX S-function that is treated identically by the Simulink engine and generated code. In general, you implement your algorithm once according to the S-function API. The Simulink engine and generated code call the S-function routines (for example, `mdlOutputs`) during model execution.

Additional memory and computation resources are required for each instance of a noninlined S-Function block. However, this routine of incorporating algorithms into models and code generation applications is typical during the prototyping phase of a project where efficiency is not important. The advantage gained by forgoing efficiency is the ability to change model parameters and structures rapidly.

Writing a noninlined S-function does not involve TLC coding. Noninlined S-functions are the default case for the build process in the sense that once you build a MEX S-function in your model, there is no additional preparation prior to pressing **Ctrl+B** to build your model.

Some restrictions exist concerning the names and locations of noninlined S-function files when generating makefiles. See "Write Noninlined S-Function and TLC Files" on page 11-66.

Inlined S-Functions

For S-functions to work in the Simulink environment, some overhead code is generated. When the code generator produces code from models that contain S-functions (without `sfunction.tlc` files), it embeds some of this overhead code in the generated code. If you want to optimize your real-time code and eliminate some of the overhead code, you must *inline* (or embed) your S-functions. This involves writing a TLC (`sfunction.tlc`) file that eliminates overhead code from the generated code. The Target Language Compiler processes `sfunction.tlc` files to define how to inline your S-function algorithm in the generated code.

Note The term *inline* should not be confused with the C++ *inline* keyword. Inline means to specify text in place of the call to the general S-function API routines (for example, `mdlOutputs`). For example, when a TLC file is used to inline an S-function, the generated code contains the C/ C++ code that would normally appear within the S-function routines and the S-function itself has been removed from the build process.

A fully inlined S-function builds your algorithm (block) into generated code in a manner that is indistinguishable from a built-in block. Typically, a fully inlined S-function requires you to implement your algorithm twice: once for the Simulink model (C/C++ MEX S-function) and once for code generation (TLC file). The complexity of the TLC file depends on the complexity of your algorithm and the level of efficiency you're trying to achieve in the generated code. TLC files vary from simple to complex in structure.

Autogenerated S-Functions for Legacy or Custom Code

If you need to invoke hand-written C/C++ code in your model, consider using the Simulink Legacy Code Tool. The Legacy Code Tool can automate the generation of a fully inlined S-function and a corresponding TLC file based on information that you register in a Legacy Code Tool data structure.

For more information, see "Integrate C Functions Using Legacy Code Tool" (Simulink) and see "Import Calls to External Code into Generated Code with Legacy Code Tool" on page 11-7.

Files Required for Implementing Noninlined and Inlined S-Functions

This topic briefly describes what files and functions you need to create noninlined and inlined S-functions.

- Noninlined S-functions require the C or C++ MEX S-function source code (*`sfunction`*`.c` or *`sfunction`*`.cpp`).
- Fully inlined S-functions require an *`sfunction`*`.tlc` file. Fully inlined S-functions produce the optimal code for a parameterized S-function. This is an S-function that operates in a specific mode dependent upon fixed S-function parameters that do not change during model execution. For a given operating mode, the *`sfunction`*`.tlc` file specifies the exact code that is generated to implement the algorithm for that mode. For example, the direct-index lookup table S-function in "Write Fully Inlined S-Functions with mdlRTW Routine" on page 11-78 contains two operating modes — one for evenly spaced `x-data` and one for unevenly spaced `x-data`.

  Note: Fully-inlined S-functions that are generated to invoke legacy or custom C/C++ code also require an *`sfunction`*`.tlc` file, which is generated by Legacy Code Tool.

Fully inlined S-functions might require the placement of the `mdlRTW` routine in your S-function MEX-file *`sfunction`*`.c` or *`sfunction`*`.cpp`. The `mdlRTW` routine lets you place information in *`model`*`.rtw`, the record file that specifies a model, and which the code generator invokes the Target Language Compiler to process prior to executing *`sfunction`*`.tlc` when generating code.

Including a `mdlRTW` routine is useful when you want to introduce nontunable parameters into your TLC file. Such parameters are generally used to determine which operating mode is active in a given instance of the S-function. Based on this information, the TLC file for the S-function can generate highly efficient, optimal code for that operating mode.

Guidelines for Writing S-Functions that Support Code Generation

- You can use C/C++ MEX, MATLAB language, and Fortran MEX S-functions with code generation.

- You can inline S-functions for code generation by providing an inlining TLC file. See S-Function Inlining in "Target Language Compiler" (Simulink Coder). MATLAB and Fortran MEX S-functions must be inlined. C/C++ MEX S-functions can be inlined for code efficiency, or noninlined.
- To automatically generate a fully inlined C MEX S-function for invoking legacy or custom code, use the Legacy Code Tool. For more information, see "Integrate C Functions Using Legacy Code Tool" (Simulink) and see "Import Calls to External Code into Generated Code with Legacy Code Tool" (Simulink Coder).
- If code efficiency is not an overriding consideration, for example, if you are rapid prototyping, you can choose not to inline a C/C++ MEX S-function. For more information, see "Write Noninlined S-Function and TLC Files" on page 11-66.

More About

- "Import Calls to External Code into Generated Code with Legacy Code Tool" on page 11-7
- "Automate S-Function Generation with S-Function Builder" on page 11-61
- "Write S-Function and TLC Files By Hand" on page 11-66

Import Calls to External Code into Generated Code with Legacy Code Tool

In this section...

Legacy Code Tool and Code Generation

You can use the Simulink Legacy Code Tool to generate fully inlined C MEX S-functions for legacy or custom code. The S-functions are optimized for embedded components, such as device drivers and lookup tables, and they call existing C or C++ functions.

Note: The Legacy Code Tool can interface with C++ functions, but not C++ objects. To work around this issue so that the tool can interface with C++ objects, see "Legacy Code Tool Limitations" (Simulink).

You can use the tool to:

- Compile and build the generated S-function for simulation.
- Generate a masked S-Function block that is configured to call the existing external code.

If you want to include these types of S-functions in models for which you intend to generate code, use the tool to generate a TLC block file. The TLC block file specifies how the generated code for a model calls the existing C or C++ function.

If the S-function depends on files in folders other than the folder containing the S-function dynamically loadable executable file, use the tool to generate an

*sFunction*_makecfg.m or rtwmakecfg.m file for the S-function. Generating the file maintains those dependencies when you build a model that includes the S-function. For example, for some applications, such as custom targets, you might want to locate files in a target-specific location. The build process looks for *sFunction*_makecfg.m or rtwmakecfg.m in the same folder as the S-function dynamically loadable executable and calls the function in the file.

For more information, see "Integrate C Functions Using Legacy Code Tool" (Simulink).

Generate Inlined S-Function Files for Code Generation

Depending on the code generation requirements of your application, to generate code for a model that uses the S-function, do either of the following:

- Generate one .cpp file for the inlined S-function. In the Legacy Code Tool data structure, set the value of the Options.singleCPPMexFile field to true before generating the S-function source file from your existing C function. For example:

  ```
  def.Options.singleCPPMexFile = true;
  legacy_code('sfcn_cmex_generate', def);
  ```

- Generate a source file and a TLC block file for the inlined S-function. For example:

  ```
  def.Options.singleCPPMexFile = false;
  legacy_code('sfcn_cmex_generate', def);
  legacy_code('sfcn_tlc_generate', def);
  ```

singleCPPMexFile Limitations

You cannot set the singleCPPMexFile field to true if

- Options.language='C++'
- You use one of the following Simulink objects with the IsAlias property set to true:
  - Simulink.Bus
  - Simulink.AliasType
  - Simulink.NumericType
- The Legacy Code Tool function specification includes a void* or void** to represent scalar work data for a state argument
- HeaderFiles field of the Legacy Code Tool structure specifies multiple header files

Apply Code Style Settings to Legacy Functions

To apply the model configuration parameters for code style to a legacy function:

1. Initialize the Legacy Code Tool data structure. For example:

   ```
   def = legacy_code('initialize');
   ```

2. In the data structure, set the value of the `Options.singleCPPMexFile` field to `true`. For example:

   ```
   def.Options.singleCPPMexFile = true;
   ```

To check the setting, enter:

```
def.Options.singleCPPMexFile
```

singleCPPMexFile Limitations

You cannot set the `singleCPPMexFile` field to `true` if

- `Options.language='C++'`
- You use one of the following Simulink objects with the `IsAlias` property set to `true`:
  - `Simulink.Bus`
  - `Simulink.AliasType`
  - `Simulink.NumericType`
- The Legacy Code Tool function specification includes a `void*` or `void**` to represent scalar work data for a state argument
- `HeaderFiles` field of the Legacy Code Tool structure specifies multiple header files

Address Dependencies on Files in Different Locations

By default, the Legacy Code Tool assumes that files on which an S-function depends reside in the same folder as the dynamically loadable executable file for the S-function. If your S-function depends on files that reside elsewhere and you are using the template makefile build process, generate an *`sFunction`*`_makecfg.m` or `rtwmakecfg.m` file for the S-function. For example, you might generate this file if your Legacy Code Tool data structure defines compilation resources as path names.

To generate the *`sFunction`*`_makecfg.m` or `rtwmakecfg.m` file, call the `legacy_code` function with `'sfcn_makecfg_generate'` or `'rtwmakecfg_generate'` as the first

argument, and the name of the Legacy Code Tool data structure as the second argument. For example:

```
legacy_code('sfcn_makecfg_generate', lct_spec);
```

If you use multiple registration files in the same folder and generate an S-function for each file with a single call to `legacy_code`, the call to `legacy_code` that specifies `'sfcn_makecfg_generate'` or `'rtwmakecfg_generate'` must be common to all registration files. For more information, see "Handling Multiple Registration Files" (Simulink) in the Simulink documentation.

For example, if you define `defs` as an array of Legacy Code Tool structures, you call `legacy_code` with `'sfcn_makecfg_generate'` once.

```
defs = [defs1(:);defs2(:);defs3(:)];
legacy_code('sfcn_makecfg_generate', defs);
```

For more information, see "Build Support for S-Functions" (Simulink Coder).

Deploy S-Functions for Simulation and Code Generation

You can deploy the S-functions that you generate with the Legacy Code Tool so that other people can use them. To deploy an S-function for simulation and code generation, share the following files:

- Registration file
- Compiled dynamically loadable executable
- TLC block file
- *`sFunction`*`_makecfg.m` or `rtwmakecfg.m` file
- Header, source, and include files on which the generated S-function depends

When you use these deployed files:

- Before using the deployed files in a Simulink model, add the folder that contains the S-function files to the MATLAB path.
- If the Legacy Code Tool data structure registers required files as absolute paths and the location of the files changes, regenerate the *`sFunction`*`_makecfg.m` or `rtwmakecfg.m` file.

Integrate External C++ Object Methods

Integrate legacy C++ object methods by using the Legacy Code Tool.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C++ MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a block TLC file and optional rtwmakecfg.m file that calls the legacy code during code generation.

Provide the Legacy Function Specification

Functions provided with the Legacy Code Tool take a specific data structure or array of structures as the argument. The data structure is initialized by calling the function legacy_code() using 'initialize' as the first input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. For detailed help on the properties, call legacy_code('help'). The definition of the legacy C++ class in this example is:

```
class adder {
  private:
      int int_state;
  public:
      adder();
      int add_one(int increment);
      int get_val();
};
```

The legacy source code is in the files adder_cpp.h and adder_cpp.cpp.

```
% rtwdemo_sfun_adder_cpp
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_adder_cpp';
def.StartFcnSpec  = 'createAdder()';
def.OutputFcnSpec = 'int32 y1 = adderOutput(int32 u1)';
def.TerminateFcnSpec = 'deleteAdder()';
def.HeaderFiles   = {'adder_cpp.h'};
def.SourceFiles   = {'adder_cpp.cpp'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
```

```
def.Options.language = 'C++';
def.Options.useTlcWithAccel = false;
```

Generate an S-Function for Simulation

To generate a C-MEX S-function according to the description provided by the input argument 'def', call the function legacy_code() again with the first input set to 'sfcn_cmex_generate'. The S-function calls the legacy functions during simulation. The source code for the S-function is in the file rtwdemo_sfun_adder_cpp.cpp.

```
legacy_code('sfcn_cmex_generate', def);
```

Compile the Generated S-Function for Simulation

After you generate the C-MEX S-function source file, to compile the S-function for simulation with Simulink®, call the function legacy_code() again with the first input set to 'compile'.

```
legacy_code('compile', def);
```

```
### Start Compiling rtwdemo_sfun_adder_cpp
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc17a_538369_5
Building with 'Microsoft Visual C++ 2013 Professional'.
MEX completed successfully.
    mex('rtwdemo_sfun_adder_cpp.cpp', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src
Building with 'Microsoft Visual C++ 2013 Professional'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_adder_cpp
### Exit
```

Generate a TLC Block File for Code Generation

After you compile the S-function and use it in simulation, you can call the function legacy_code() again. Set the first input to 'sfcn_tlc_generate' to generate a TLC block file that supports code generation through Simulink® Coder™. If the TLC block file is not created and you try to generate code for a model that includes the S-function, code generation fails. The TLC block file for the S-function is: rtwdemo_sfun_adder_cpp.tlc.

```
legacy_code('sfcn_tlc_generate', def);
```

Generate an rtwmakecfg.m File for Code Generation

After you create the TLC block file, you can call the function legacy_code() again. Set the first input to 'rtwmakecfg_generate' to generate an rtwmakecfg.m file that supports

code generation through Simulink® Coder™. If the required source and header files for the S-function are not in the same folder as the S-function, and you want to add these dependencies in the makefile produced during code generation, generate the rtwmakecfg.m file.

```
legacy_code('rtwmakecfg_generate', def);
```

Generate a Masked S-Function Block for Calling the Generated S-Function

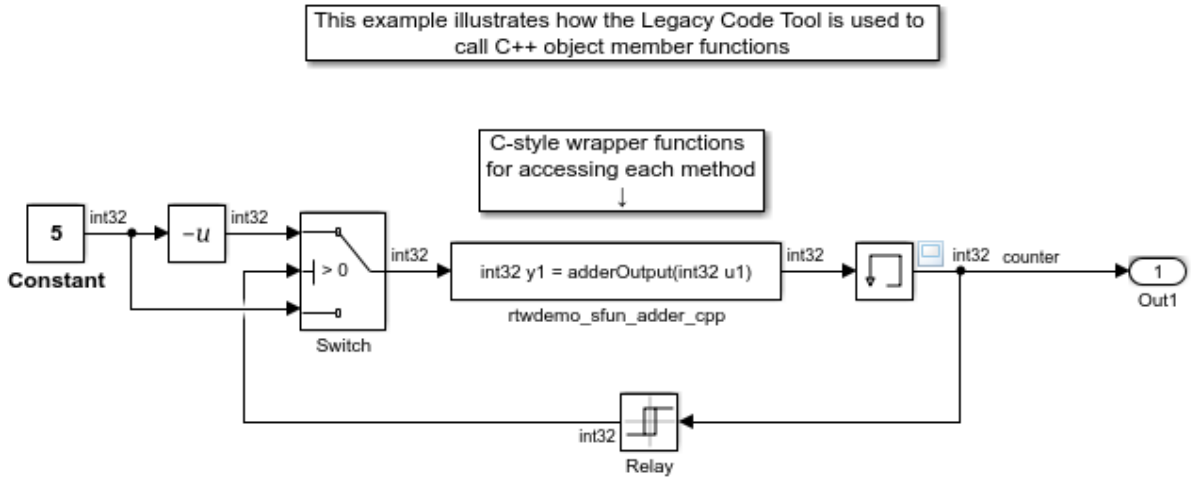
After you compile the C-MEX S-function source, you can call the function legacy_code() again. Set the first input to 'slblock_generate' to generate a masked S-function block that is configured to call that S-function. The software places the block in a new model. You can copy the block to an existing model.

```
% legacy_code('slblock_generate', def);
```

Show the Generated Integration with Legacy Code

The model rtwdemo_lct_cpp shows integration with the legacy code.

```
open_system('rtwdemo_lct_cpp')
sim('rtwdemo_lct_cpp')
```



Integrate External C++ Objects

The Legacy Code Tool can interface with C++ functions, but not C++ objects. Using the previous example as a starting point, here is an example of how you can work around this limitation.

- Modify the class definition for `adder` in a new file `adder_cpp.hpp`. Add three new macros that dynamically allocate a new `adder` object, invoke the method `add_one()`, and free the memory allocated. Each macro takes a pointer to an `adder` object. Because each function called by the Legacy Code Tool must have a C-like signature,

the pointer is cached and passed as a `void*`. Then you must explicitly cast to `adder*` in the macro. The new class definition for `adder`:

```
#ifndef _ADDER_CPP_
#define _ADDER_CPP_

class adder {
private:
 int int_state;
public:
 adder(): int_state(0) {};
 int add_one(int increment);
 int get_val() {return int_state;};
};

// Method wrappers implemented as macros
#define createAdder(work1) \
    *(work1) = new adder

#define deleteAdder(work1) \
    delete(static_cast(adder*)(*(work1)))

#define adderOutput(work1, u1) \
    (static_cast(adder*) ((work1)))->add_one(u1)

#endif /* _ADDER_CPP_ */
```

- Update `adder_cpp.cpp`. With the class modification, instead of one global instance, each generated S-function manages its own `adder` object.

```
#include "adder_cpp.hpp"

int adder::add_one(int increment)
{
 int_state += increment;
    return int_state;
}
```

- Update `rtwdemo_sfun_adder_cpp.cpp` with the following changes:

  - `StartFcnSpec` calls the macro that allocates a new `adder` object and caches the pointer.

    ```
    def.StartFcnSpec  = 'createAdder(void **work1)';
    ```

- `OutputFcnSpec` calls the macro that invokes the method `add_one()` and provides the S-function specific `adder` pointer object.

  ```
  def.OutputFcnSpec = 'int32 y1 = adderOutput(void *work1, int32 u1)';
  ```

- `TerminateFcnSpec` calls the macro that frees the memory.

  ```
  def.TerminateFcnSpec = 'deleteAdder(void **work1)';
  ```

Legacy Code Tool Examples

Integrate External C Functions That Pass Input Arguments By Value Versus Address

This example shows how to use the Legacy Code Tool to integrate legacy C functions that pass their input arguments by value versus address.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C-MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a TLC block file and optional rtwmakecfg.m file that specifies how the generated code for a model calls the legacy code.

Provide the Legacy Function Specification

Legacy Code Tool functions take a specific data structure or array of structures as the argument. You can initialize the data structure by calling the function legacy_code() using 'initialize' as the first input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. For detailed help on the properties, call legacy_code('help'). The prototypes of the legacy functions being called in this example are:

- FLT filterV1(const FLT signal, const FLT prevSignal, const FLT gain)
- FLT filterV2(const FLT* signal, const FLT prevSignal, const FLT gain)

FLT is a typedef to float. The legacy source code is in the files your_types.h, myfilter.h, filterV1.c, and filterV2.c.

Note the difference in the OutputFcnSpec defined in the two structures; the first case specifies that the first input argument is passed by value, while the second case specifies pass by pointer.

```
defs = [];

% rtwdemo_sfun_filterV1
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_filterV1';
def.OutputFcnSpec = 'single y1 = filterV1(single u1, single u2, single p1)';
def.HeaderFiles   = {'myfilter.h'};
def.SourceFiles   = {'filterV1.c'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
defs = [defs; def];

% rtwdemo_sfun_filterV2
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_filterV2';
def.OutputFcnSpec = 'single y1 = filterV2(single u1[1], single u2, single p1)';
def.HeaderFiles   = {'myfilter.h'};
def.SourceFiles   = {'filterV2.c'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
defs = [defs; def];
```

Generate S-Functions for Simulation

To generate C-MEX S-functions according to the description provided by the input argument 'defs', call the function legacy_code() again with the first input set to 'sfcn_cmex_generate'. The S-functions call the legacy functions in simulation. The source code for the S-functions is in the files rtwdemo_sfun_filterV1.c and rtwdemo_sfun_filterV2.c.

```
legacy_code('sfcn_cmex_generate', defs);
```

Compile the Generated S-Functions for Simulation

After you generate the C-MEX S-function source files, to compile the S-functions for simulation with Simulink®, call the function legacy_code() again with the first input set to 'compile'.

```
legacy_code('compile', defs);
```

```
### Start Compiling rtwdemo_sfun_filterV1
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc17a_538369_5
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
```

```
MEX completed successfully.
    mex('rtwdemo_sfun_filterV1.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src',
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_filterV1
### Exit

### Start Compiling rtwdemo_sfun_filterV2
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc17a_538369_5
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_filterV2.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src',
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_filterV2
### Exit
```

Generate TLC Block Files for Code Generation

After you compile the S-functions and use them in simulation, you can call the function legacy_code() again with the first input set to 'sfcn_tlc_generate' to generate TLC block files. Block files specify how the generated code for a model calls the legacy code. If you do not generate TLC block files and you try to generate code for a model that includes the S-functions, code generation fails. The TLC block files for the S-functions are rtwdemo_sfun_filterV1.tlc and rtwdemo_sfun_filterV2.tlc.

```
legacy_code('sfcn_tlc_generate', defs);
```

Generate an rtwmakecfg.m File for Code Generation

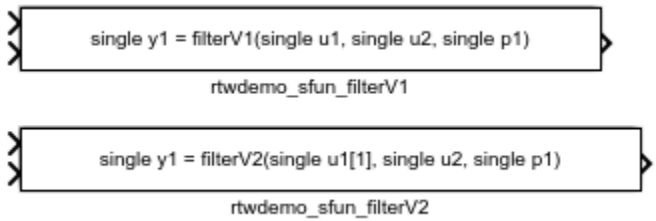
After you create the TLC block files, you can call the function legacy_code() again with the first input set to 'rtwmakecfg_generate' to generate an rtwmakecfg.m file to support code generation. If the required source and header files for the S-functions are not in the same folder as the S-functions, and you want to add these dependencies in the makefile produced during code generation, generate the rtwmakecfg.m file.

```
legacy_code('rtwmakecfg_generate', defs);
```

Generate Masked S-Function Blocks for Calling the Generated S-Functions

After you compile the C-MEX S-function source, you can call the function legacy_code() again with the first input set to 'slblock_generate' to generate masked S-function blocks that call the S-functions. The software places the blocks in a new model. From there you can copy them to an existing model.

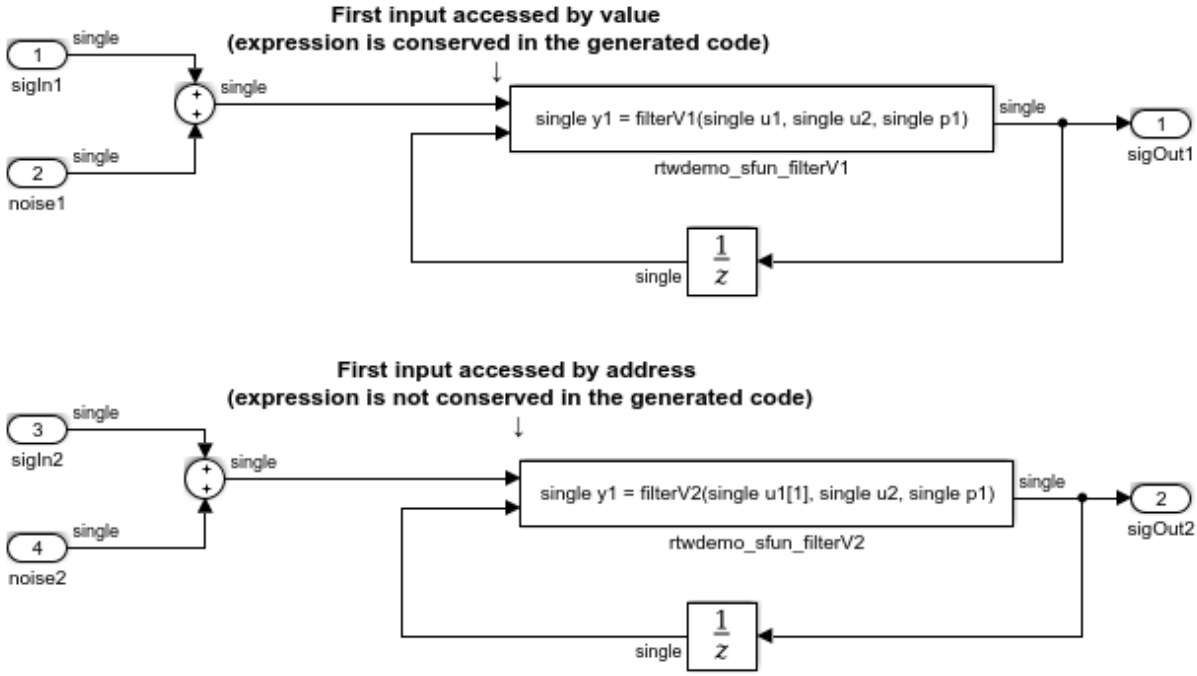
```
legacy_code('slblock_generate', defs);
```



Show the Generated Integration with Legacy Code

The model rtwdemo_lct_filter shows integration of the model with the legacy code. The subsystem TestFilter serves as a harness for the calls to the legacy C functions via the generate S-functions, with unit delays serving to store the previous output values.

```
open_system('rtwdemo_lct_filter')
open_system('rtwdemo_lct_filter/TestFilter')
sim('rtwdemo_lct_filter')
```



Integrate External C Functions That Pass the Output Argument As a Return Argument

This example shows how to use the Legacy Code Tool to integrate legacy C functions that pass their output as a return argument.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C-MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a TLC block file and optional rtwmakecfg.m file that specifies how the generated code for a model calls the legacy code.

Provide the Legacy Function Specification

Legacy Code Tool functions take a specific data structure or array of structures as the argument. You can initialize the data structure by calling the function legacy_code()

using 'initialize' as the first input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. For detailed help on the properties, call legacy_code('help'). The prototype of the legacy functions being called in this example is:

FLT gainScalar(const FLT in, const FLT gain)

FLT is a typedef to float. The legacy source code is in the files your_types.h, gain.h, and gainScalar.c.

```
% rtwdemo_sfun_gain_scalar
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_gain_scalar';
def.OutputFcnSpec = 'single y1 = gainScalar(single u1, single p1)';
def.HeaderFiles   = {'gain.h'};
def.SourceFiles   = {'gainScalar.c'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
```

Generate an S-Function for Simulation

To generate a C-MEX S-function according to the description provided by the input argument 'def', call the function legacy_code() again with the first input set to 'sfcn_cmex_generate'. The S-function calls the legacy functions during simulation. The source code for the S-function is in the file rtwdemo_sfun_gain_scalar.c.

```
legacy_code('sfcn_cmex_generate', def);
```

Compile the Generated S-Function for Simulation

After you generate the C-MEX S-function source file, to compile the S-function for simulation with Simulink®, call the function legacy_code() again with the first input set to 'compile'.

```
legacy_code('compile', def);
```

```
### Start Compiling rtwdemo_sfun_gain_scalar
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc17a_538369_5
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_gain_scalar.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
```

```
### Finish Compiling rtwdemo_sfun_gain_scalar
### Exit
```

Generate a TLC Block File for Code Generation

After you compile the S-function and use it in simulation, you can call the function legacy_code() again with the first input set to 'sfcn_tlc_generate' to generate a TLC block file. The block file specifies how the generated code for a model calls the legacy code. If you do not generate a TLC block file and you try to generate code for a model that includes the S-function, code generation fails. The TLC block file for the S-function is: rtwdemo_sfun_gain_scalar.tlc.

```
legacy_code('sfcn_tlc_generate', def);
```

Generate an rtwmakecfg.m File for Code Generation

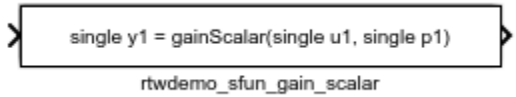
After you create the TLC block file, you can call the function legacy_code() again with the first input set to 'rtwmakecfg_generate' to generate an rtwmakecfg.m file to support code generation. If the required source and header files for the S-function are not in the same folder as the S-function, and you want to add these dependencies in the makefile produced during code generation, generate the rtwmakecfg.m file.

```
legacy_code('rtwmakecfg_generate', def);
```

Generate a Masked S-Function Block for Calling the Generated S-Function

After you compile the C-MEX S-function source, you can call the function legacy_code() again with the first input set to 'slblock_generate' to generate a masked S-function block that calls that S-function. The software places the block in a new model. From there you can copy it to an existing model.

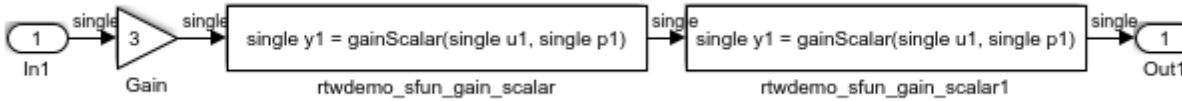
```
legacy_code('slblock_generate', def);
```



Show the Generated Integration with Legacy Code

The model rtwdemo_lct_gain shows integration of the model with the legacy code. The subsystem TestGain serves as a harness for the call to the legacy C function via the generate S-function.

```
open_system('rtwdemo_lct_gain')
open_system('rtwdemo_lct_gain/TestGain')
sim('rtwdemo_lct_gain')
```



Integrate External C Functions That Pass Input and Output Arguments as Signals with a Fixed-Point Data Type

This example shows how to use the Legacy Code Tool to integrate legacy C functions that pass their inputs and outputs by using parameters of fixed-point data type.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C-MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a TLC block file and optional rtwmakecfg.m file that specifies how the generated code for a model calls the legacy code.

Provide the Legacy Function Specification

Legacy Code Tool functions take a specific data structure or array of structures as the argument. You can initialize the data structure by calling the function legacy_code() using 'initialize' as the first input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. For detailed help on the properties, call legacy_code('help'). The prototype of the legacy functions being called in this example is:

myFixpt timesS16(const myFixpt in1, const myFixpt in2, const uint8_T fracLength)

myFixpt is logically a fixed-point data type, which is physically a typedef to a 16-bit integer:

```
myFixpt = Simulink.NumericType;
myFixpt.DataTypeMode = 'Fixed-point: binary point scaling';
```

```
myFixpt.Signed = true;
myFixpt.WordLength = 16;
myFixpt.FractionLength = 10;
myFixpt.IsAlias = true;
myFixpt.HeaderFile = 'timesFixpt.h';
```

The legacy source code is in the files timesFixpt.h, and timesS16.c.

```
% rtwdemo_sfun_times_s16
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_times_s16';
def.OutputFcnSpec = 'myFixpt y1 = timesS16(myFixpt u1, myFixpt u2, uint8 p1)';
def.HeaderFiles   = {'timesFixpt.h'};
def.SourceFiles   = {'timesS16.c'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
```

Generate an S-Function for Simulation

To generate a C-MEX S-function according to the description provided by the input argument 'def', call the function legacy_code() again with the first input set to 'sfcn_cmex_generate'. The S-function calls the legacy functions during simulation. The source code for the S-function is in the file rtwdemo_sfun_times_s16.c.

```
legacy_code('sfcn_cmex_generate', def);
```

Compile the Generated S-Function for Simulation

After you generate the C-MEX S-function source file, to compile the S-function for simulation with Simulink®, call the function legacy_code() again with the first input set to 'compile'.

```
legacy_code('compile', def);
```

```
### Start Compiling rtwdemo_sfun_times_s16
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc17a_538369_5
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_times_s16.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src'
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_times_s16
### Exit
```

Generate a TLC Block File for Code Generation

After you compile the S-function and use it in simulation, you can call the function legacy_code() again with the first input set to 'sfcn_tlc_generate' to generate a TLC block file. The block file specifies how the generated code for a model calls the legacy code. If you do not generate a TLC block file and you try to generate code for a model that includes the S-function, code generation fails. The TLC block file for the S-function is: rtwdemo_sfun_times_s16.tlc.

```
legacy_code('sfcn_tlc_generate', def);
```

Generate an rtwmakecfg.m File for Code Generation

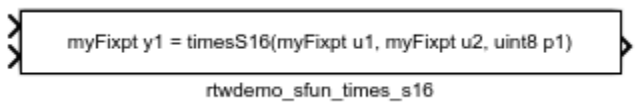
After you create the TLC block file, you can call the function legacy_code() again with the first input set to 'rtwmakecfg_generate' to generate an rtwmakecfg.m file to support code generation. If the required source and header files for the S-function are not in the same folder as the S-function, and you want to add these dependencies in the makefile produced during code generation, generate the rtwmakecfg.m file.

```
legacy_code('rtwmakecfg_generate', def);
```

Generate a Masked S-Function Block for Calling the Generated S-Function

After you compile the C-MEX S-function source, you can call the function legacy_code() again with the first input set to 'slblock_generate' to generate a masked S-function block that calls that S-function. The software places the block in a new model. From there you can copy it to an existing model.

```
legacy_code('slblock_generate', def);
```

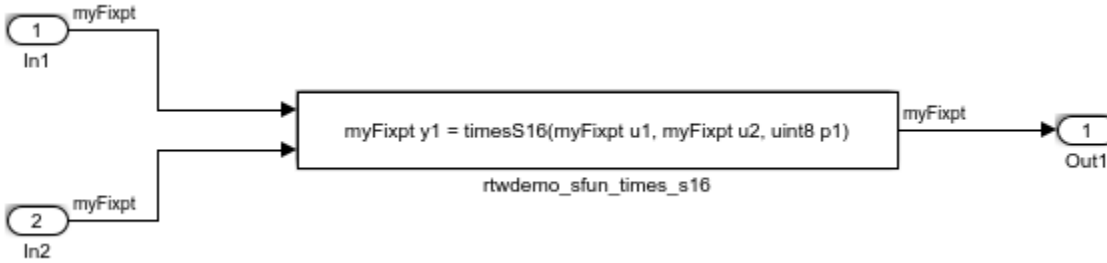


Show the Integration of the Model with Legacy Code

The model rtwdemo_lct_fixpt_signals shows integration of the model with the legacy code. The subsystem TestFixpt serves as a harness for the call to the legacy C function via the generated S-function.

```
open_system('rtwdemo_lct_fixpt_signals')
```

```
open_system('rtwdemo_lct_fixpt_signals/TestFixpt')
sim('rtwdemo_lct_fixpt_signals')
```



Integrate External C Functions with Instance-Specific Persistent Memory

Integrate legacy C functions that use instance-specific persistent memory by using the Legacy Code Tool.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C-MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a TLC block file and optional rtwmakecfg.m file that specifies how the generated code for a model calls the legacy code.

Provide the Legacy Function Specification

Legacy Code Tool functions take a specific data structure or array of structures as the argument. You can initialize the data structure by calling the function legacy_code() using 'initialize' as the first input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. For detailed help on the properties, call legacy_code('help'). The prototypes of the legacy functions being called in this example are:

void memory_bus_init(COUNTERBUS *mem, int32_T upper_sat, int32_T lower_sat);

void memory_bus_step(COUNTERBUS *input, COUNTERBUS *mem, COUNTERBUS *output);

mem is an instance-specific persistent memory for applying a one integration step delay. COUNTERBUS is a struct typedef defined in counterbus.h and implemented with a Simulink.Bus object in the base workspace. The legacy source code is in the files memory_bus.h, and memory_bus.c.

```
evalin('base','load rtwdemo_lct_data.mat')

% rtwdemo_sfun_work
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_work';
def.InitializeConditionsFcnSpec = ...
    'void memory_bus_init(COUNTERBUS work1[1], int32 p1, int32 p2)';
def.OutputFcnSpec = ...
    'void memory_bus_step(COUNTERBUS u1[1], COUNTERBUS work1[1], COUNTERBUS y1[1])';
def.HeaderFiles   = {'memory_bus.h'};
def.SourceFiles   = {'memory_bus.c'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
```

Generate an S-Function for Simulation

To generate a C-MEX S-function according to the description provided by the input argument 'def', call the function legacy_code() again with the first input set to 'sfcn_cmex_generate'. The S-function calls the legacy functions during simulation. The source code for the S-function is in the file rtwdemo_sfun_work.c.

```
legacy_code('sfcn_cmex_generate', def);
```

Compile the Generated S-Function for Simulation

After you generate the C-MEX S-function source file, to compile the S-function for simulation with Simulink®, call the function legacy_code() again with the first input set to 'compile'.

```
legacy_code('compile', def);
```

```
### Start Compiling rtwdemo_sfun_work
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc17a_538369_5
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_work.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
```

```
### Finish Compiling rtwdemo_sfun_work
### Exit
```

Generate a TLC Block File for Code Generation

After you compile the S-function and use it in simulation, you can call the function legacy_code() again with the first input set to 'sfcn_tlc_generate' to generate a TLC block file. The block file specifies how the generated code for a model calls the legacy code. If you do not generate a TLC block file and you try to generate code for a model that includes the S-function, code generation fails. The TLC block file for the S-function is: rtwdemo_sfun_work.tlc.

```
legacy_code('sfcn_tlc_generate', def);
```

Generate an rtwmakecfg.m File for Code Generation

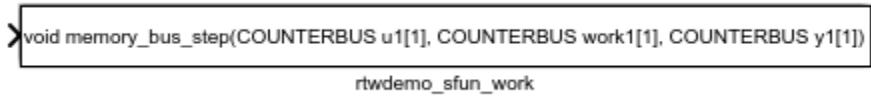
After you create the TLC block file, you can call the function legacy_code() again with the first input set to 'rtwmakecfg_generate' to generate an rtwmakecfg.m file to support code generation. If the required source and header files for the S-function are not in the same folder as the S-function, and you want to add these dependencies in the makefile produced during code generation, generate the rtwmakecfg.m file.

```
legacy_code('rtwmakecfg_generate', def);
```

Generate a Masked S-Function Block for Calling the Generated S-Function

After you compile the C-MEX S-function source, you can call the function legacy_code() again with the first input set to 'slblock_generate' to generate a masked S-function block that calls that S-function. The software places the block in a new model. From there you can copy it to an existing model.

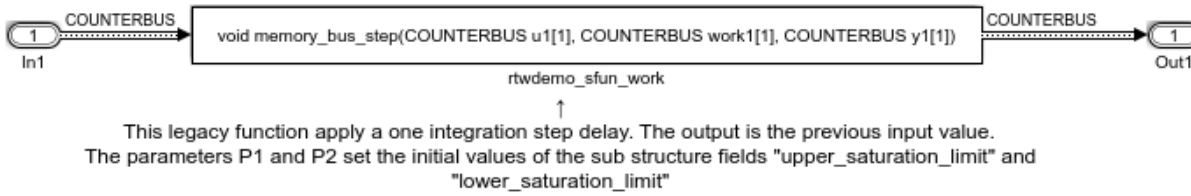
```
legacy_code('slblock_generate', def);
```



Show the Integration of the Model with Legacy Code

The model rtwdemo_lct_work shows integration of the model with the legacy code. The subsystem memory_bus serves as a harness for the call to the legacy C function.

```
open_system('rtwdemo_lct_work')
open_system('rtwdemo_lct_work/memory_bus')
sim('rtwdemo_lct_work')
```



Integrate External C Functions That Use Structure Arguments

Integrate legacy C functions with structure arguments that use Simulink® buses with the Legacy Code Tool.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C-MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a TLC block file and optional rtwmakecfg.m file that specifies how the generated code for a model calls the legacy code.

Provide the Legacy Function Specification

Legacy Code Tool functions take a specific data structure or array of structures as the argument. You can initialize the data structure by calling the function legacy_code() using 'initialize' as the first input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. For detailed help on the properties, call legacy_code('help'). The prototype of the legacy functions being called in this example is:

counterbusFcn(COUNTERBUS *u1, int32_T u2, COUNTERBUS *y1, int32_T *y2)

COUNTERBUS is a struct typedef defined in counterbus.h and implemented with a Simulink.Bus object in the base workspace. The legacy source code is in the files counterbus.h, and counterbus.c.

```
evalin('base','load rtwdemo_lct_data.mat')

% rtwdemo_sfun_counterbus
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_counterbus';
def.OutputFcnSpec = ...
    'void counterbusFcn(COUNTERBUS u1[1], int32 u2, COUNTERBUS y1[1], int32 y2[1])';
def.HeaderFiles   = {'counterbus.h'};
def.SourceFiles   = {'counterbus.c'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
```

Generate an S-Function for Simulation

To generate a C-MEX S-function according to the description provided by the input argument 'def', call the function legacy_code() again with the first input set to 'sfcn_cmex_generate'. The S-function calls the legacy functions during simulation. The source code for the S-function is in the file rtwdemo_sfun_counterbus.c.

```
legacy_code('sfcn_cmex_generate', def);
```

Compile the Generated S-Function for Simulation

After you generate the C-MEX S-function source file, to compile the S-function for simulation with Simulink®, call the function legacy_code() again with the first input set to 'compile'.

```
legacy_code('compile', def);
```

```
### Start Compiling rtwdemo_sfun_counterbus
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc17a_538369_5
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_counterbus.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_counterbus
### Exit
```

Generate a TLC Block File for Code Generation

After you compile the S-function and use it in simulation, you can call the function legacy_code() again with the first input set to 'sfcn_tlc_generate' to generate a TLC block

file. The block file specifies how the generated code for a model calls the legacy code. If you do not generate a TLC block file and you try to generate code for a model that includes the S-function, code generation fails. The TLC block file for the S-function is: rtwdemo_sfun_counterbus.tlc.

```
legacy_code('sfcn_tlc_generate', def);
```

Generate an rtwmakecfg.m File for Code Generation

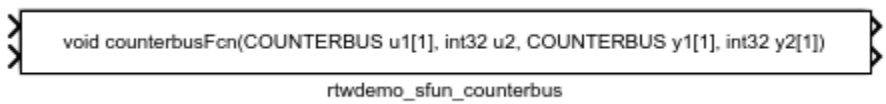
After you create the TLC block file, you can call the function legacy_code() again with the first input set to 'rtwmakecfg_generate' to generate an rtwmakecfg.m file to support code generation. If the required source and header files for the S-function are not in the same folder as the S-function, and you want to add these dependencies in the makefile produced during code generation, generate the rtwmakecfg.m file.

```
legacy_code('rtwmakecfg_generate', def);
```

Generate a Masked S-Function Block for Calling the Generated S-Function

After you compile the C-MEX S-function source, you can call the function legacy_code() again with the first input set to 'slblock_generate' to generate a masked S-function block that calls that S-function. The software places the block in a new model. From there you can copy it to an existing model.

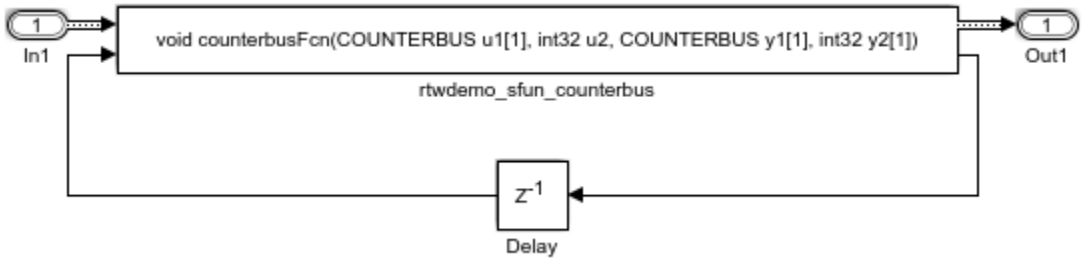
```
legacy_code('slblock_generate', def);
```



Show the Integration of the Model with Legacy Code

The model rtwdemo_lct_bus shows integration of the model with the legacy code. The subsystem TestCounter serves as a harness for the call to the legacy C function.

```
open_system('rtwdemo_lct_bus')
open_system('rtwdemo_lct_bus/TestCounter')
sim('rtwdemo_lct_bus')
```



Integrate External C Functions That Pass Input and Output Arguments as Signals with Complex Data

Integrate legacy C functions using complex signals with the Legacy Code Tool.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C-MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a TLC block file and optional rtwmakecfg.m file that specifies how the generated code for a model calls the legacy code.

Provide the Legacy Function Specification

Legacy Code Tool functions take a specific data structure or array of structures as the argument. You can initialize the data structure by calling the function legacy_code() using 'initialize' as the first input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. For detailed help on the properties, call legacy_code('help'). The prototype of the legacy functions being called in this example is:

void cplx_gain(creal_T *input, creal_T *gain, creal_T *output);

creal_T is the complex representation of a double. The legacy source code is in the files cplxgain.h, and cplxgain.c.

```
% rtwdemo_sfun_gain_scalar
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_cplx_gain';
def.OutputFcnSpec = ...
```

```
    'void cplx_gain(complex<double> u1[1], complex<double> p1[1], complex<double> y1[1]
def.HeaderFiles   = {'cplxgain.h'};
def.SourceFiles   = {'cplxgain.c'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
```

Generate an S-Function for Simulation

To generate a C-MEX S-function according to the description provided by the input argument 'def', call the function legacy_code() again with the first input set to 'sfcn_cmex_generate'. The S-function calls the legacy functions during simulation. The source code for the S-function is in the file rtwdemo_sfun_cplx_gain.c.

```
legacy_code('sfcn_cmex_generate', def);
```

Compile the Generated S-Function for Simulation

After you generate the C-MEX S-function source file, to compile the S-function for simulation with Simulink®, call the function legacy_code() again with the first input set to 'compile'.

```
legacy_code('compile', def);
```

```
### Start Compiling rtwdemo_sfun_cplx_gain
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc17a_538369_5
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_cplx_gain.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src'
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_cplx_gain
### Exit
```

Generate a TLC Block File for Code Generation

After you compile the S-function and use it in simulation, you can call the function legacy_code() again with the first input set to 'sfcn_tlc_generate' to generate a TLC block file. The block file specifies how the generated code for a model calls the legacy code. If you do not generate a TLC block file and you try to generate code for a model that includes the S-function, code generation fails. The TLC block file for the S-function is: rtwdemo_sfun_cplx_gain.tlc.

```
legacy_code('sfcn_tlc_generate', def);
```

Generate an rtwmakecfg.m File for Code Generation

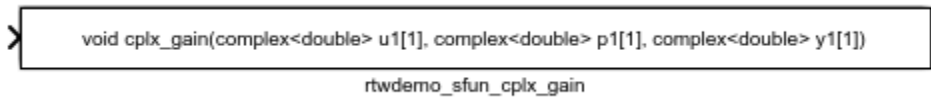
After you create the TLC block file, you can call the function legacy_code() again with the first input set to 'rtwmakecfg_generate' to generate an rtwmakecfg.m file to support code generation. If the required source and header files for the S-function are not in the same folder as the S-function, and you want to add these dependencies in the makefile produced during code generation, generate the rtwmakecfg.m file.

```
legacy_code('rtwmakecfg_generate', def);
```

Generate a Masked S-Function Block for Calling the Generated S-Function

After you compile the C-MEX S-function source, you can call the function legacy_code() again with the first input set to 'slblock_generate' to generate a masked S-function block that calls that S-function. The software places the block in a new model. From there you can copy it to an existing model.

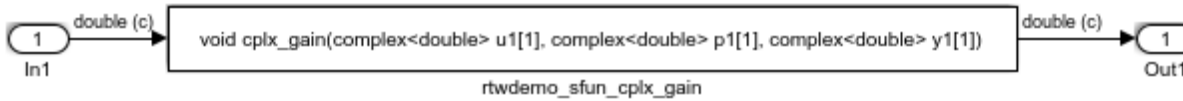
```
legacy_code('slblock_generate', def);
```



Show the Integration of the Model with Legacy Code

The model rtwdemo_lct_cplxgain shows integration of the model with the legacy code. The subsystem complex_gain serves as a harness for the call to the legacy C function via the generate S-function.

```
if isempty(find_system('SearchDepth',0,'Name','rtwdemo_lct_cplxgain'))
    open_system('rtwdemo_lct_cplxgain')
    open_system('rtwdemo_lct_cplxgain/complex_gain')
    sim('rtwdemo_lct_cplxgain')
end
```



Integrate External C Functions That Pass Arguments That Have Inherited Dimensions

This example shows how to use the Legacy Code Tool to integrate legacy C functions whose arguments have inherited dimensions.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C-MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a TLC block file and optional rtwmakecfg.m file that specifies how the generated code for a model calls the legacy code.

Provide the Legacy Function Specification

Legacy Code Tool functions take a specific data structure or array of structures as the argument. You can initialize the data structure by calling the function legacy_code() using 'initialize' as the first input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. For detailed help on the properties, call legacy_code('help'). The prototypes of the legacy functions being called in this example are:

- void mat_add(real_T *u1, real_T *u2, int32_T nbRows, int32_T nbCols, real_T *y1)
- void mat_mult(real_T *u1, real_T *u2, int32_T nbRows1, int32_T nbCols1, int32_T nbCols2, real_T *y1)

real_T is a typedef to double, and int32_T is a typedef to a 32-bit integer. The legacy source code is in the files mat_ops.h, and mat_ops.c.

```
defs = [];

% rtwdemo_sfun_mat_add
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_mat_add';
```

```
def.OutputFcnSpec = ['void mat_add(double u1[][], double u2[][], ' ...
                     'int32 u3, int32 u4, double y1[size(u1,1)][size(u1,2)])'];
def.HeaderFiles   = {'mat_ops.h'};
def.SourceFiles   = {'mat_ops.c'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
defs = [defs; def];

% rtwdemo_sfun_mat_mult
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_mat_mult';
def.OutputFcnSpec =  ['void mat_mult(double u1[p1][p2], double u2[p2][p3], '...
                     'int32 p1, int32 p2, int32 p3, double y1[p1][p3])'];
def.HeaderFiles   = {'mat_ops.h'};
def.SourceFiles   = {'mat_ops.c'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
defs = [defs; def];
```

Generate S-Functions for Simulation

To generate C-MEX S-functions according to the description provided by the input argument 'defs', call the function legacy_code() again with the first input set to 'sfcn_cmex_generate'. The S-functions call the legacy functions during simulation. The source code for the S-function is in the files rtwdemo_sfun_mat_add.c and rtwdemo_sfun_mat_mult.c.

```
legacy_code('sfcn_cmex_generate', defs);
```

Compile the Generated S-Functions for Simulation

After you generate the C-MEX S-function source files, to compile the S-functions for simulation with Simulink®, call the function legacy_code() again with the first input set to 'compile'.

```
legacy_code('compile', defs);
```

```
### Start Compiling rtwdemo_sfun_mat_add
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc17a_538369_5
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_mat_add.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src',
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
```

```
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_mat_add
### Exit

### Start Compiling rtwdemo_sfun_mat_mult
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc17a_538369_5
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_mat_mult.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src',
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_mat_mult
### Exit
```

Generate TLC Block Files for Code Generation

After you compile the S-functions and use them in simulation, you can call the function legacy_code() again with the first input set to 'sfcn_tlc_generate' to generate TLC block files. Block files specify how the generated code for a model calls the legacy code. If you do not generate TLC block files and you try to generate code for a model that includes the S-functions, code generation fails. The TLC block files for the S-functions are rtwdemo_sfun_mat_add.tlc and rtwdemo_sfun_mat_mult.tlc.

```
legacy_code('sfcn_tlc_generate', defs);
```

Generate an rtwmakecfg.m File for Code Generation

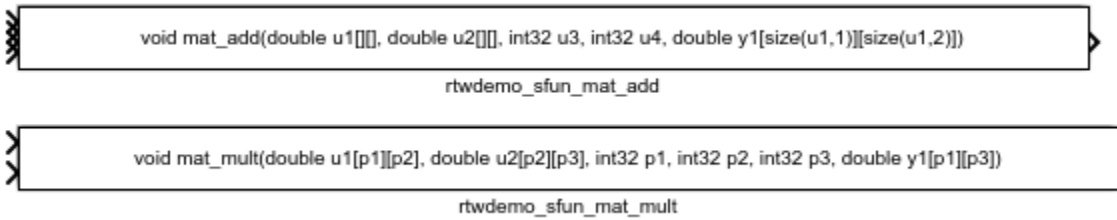
After you create the TLC block files, you can call the function legacy_code() again with the first input set to 'rtwmakecfg_generate' to generate an rtwmakecfg.m file to support code generation. If the required source and header files for the S-functions are not in the same folder as the S-functions, and you want to add these dependencies in the makefile produced during code generation, generate the rtwmakecfg.m file.

```
legacy_code('rtwmakecfg_generate', defs);
```

Generate Masked S-Function Blocks for Calling the Generated S-Functions

After you compile the C-MEX S-function source, you can call the function legacy_code() again with the first input set to 'slblock_generate' to generate masked S-function blocks that call the S-functions. The software places the blocks in a new model. From there you can copy them to an existing model

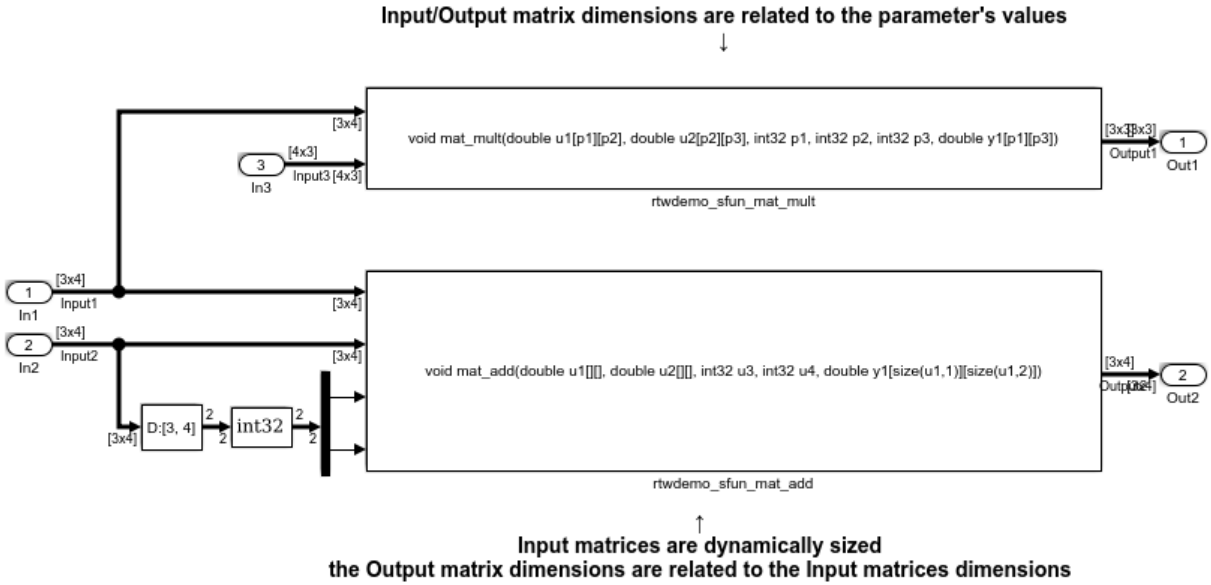
```
legacy_code('slblock_generate', defs);
```



Show the Generated Integration with Legacy Code

The model rtwdemo_lct_inherit_dims shows integration of the model with the legacy code. The subsystem TestMatOps serves as a harness for the calls to the legacy C functions, with unit delays serving to store the previous output values.

```
open_system('rtwdemo_lct_inherit_dims')
open_system('rtwdemo_lct_inherit_dims/TestMatOps')
sim('rtwdemo_lct_inherit_dims')
```



Integrate External C Functions That Implement Start and Terminate Actions

Integrate legacy C functions that have start and terminate actions by using the Legacy Code Tool.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C-MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a TLC block file and optional rtwmakecfg.m file that specifies how the generated code for a model calls the legacy code.

Provide the Legacy Function Specification

Legacy Code Tool functions take a specific data structure or array of structures as the argument. You can initialize the data structure by calling the function legacy_code() using 'initialize' as the first input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. For detailed help on the

properties, call legacy_code('help'). The prototypes of the legacy functions being called in this example are:

- void initFaultCounter(unsigned int *counter)
- void openLogFile(void **fid)
- void incAndLogFaultCounter(void *fid, unsigned int *counter, double time)
- void closeLogFile(void **fid)

The legacy source code is in the files your_types.h, fault.h, and fault.c.

```
% rtwdemo_sfun_fault
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_fault';
def.InitializeConditionsFcnSpec  = 'initFaultCounter(uint32 work2[1])';
def.StartFcnSpec                 = 'openLogFile(void **work1)';
def.OutputFcnSpec                = ...
    'incAndLogFaultCounter(void *work1, uint32 work2[1], double u1)';
def.TerminateFcnSpec             = 'closeLogFile(void **work1)';
def.HeaderFiles                  = {'fault.h'};
def.SourceFiles                  = {'fault.c'};
def.IncPaths                     = {'rtwdemo_lct_src'};
def.SrcPaths                     = {'rtwdemo_lct_src'};
def.Options.useTlcWithAccel      = false;
```

Generate an S-Function for Simulation

To generate a C-MEX S-function according to the description provided by the input argument 'def', call the function legacy_code() again with the first input set to 'sfcn_cmex_generate'. The S-function calls the legacy functions during simulation. The source code for the S-function is in the file rtwdemo_sfun_fault.c.

```
legacy_code('sfcn_cmex_generate', def);
```

Compile the Generated S-Function for Simulation

After you generate the C-MEX S-function source file, to compile the S-function for simulation with Simulink®, call the function legacy_code() again with the first input set to 'compile'.

```
legacy_code('compile', def);


### Start Compiling rtwdemo_sfun_fault
```

```
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc17a_538369_5
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_fault.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-I
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_fault
### Exit
```

Generate a TLC Block File for Code Generation

After you compile the S-function and use it in simulation, you can call the function legacy_code() again with the first input set to 'sfcn_tlc_generate' to generate a TLC block file. The block file specifies how the generated code for a model calls the legacy code. If you do not generate a TLC block file and you try to generate code for a model that includes the S-function, code generation fails. The TLC block file for the S-function is: rtwdemo_sfun_fault.tlc.

```
legacy_code('sfcn_tlc_generate', def);
```

Generate an rtwmakecfg.m File for Code Generation

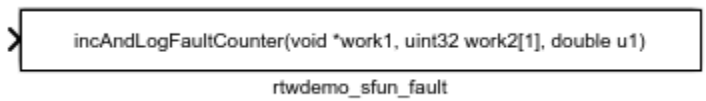
After you create the TLC block file, you can call the function legacy_code() again with the first input set to 'rtwmakecfg_generate' to generate an rtwmakecfg.m file to support code generation. If the required source and header files for the S-function are not in the same folder as the S-function, and you want to add these dependencies in the makefile produced during code generation, generate the rtwmakecfg.m file.

```
legacy_code('rtwmakecfg_generate', def);
```

Generate a Masked S-Function Block for Calling the Generated S-Function

After you compile the C-MEX S-function source, you can call the function legacy_code() again with the first input set to 'slblock_generate' to generate a masked S-function block that calls that S-function. The software places the block in a new model. From there you can copy it to an existing model.

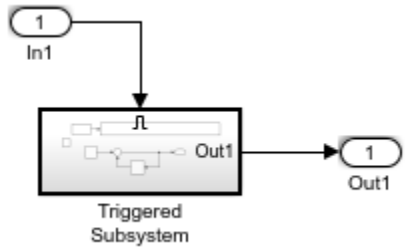
```
legacy_code('slblock_generate', def);
```



Showing the Generated Integration with Legacy Code

The model rtwdemo_lct_start_term shows integration of the model with the legacy code. The subsystem TestFixpt serves as a harness for the call to the legacy C function, and the scope compares the output of the function with the output of the built-in Simulink® product block; the results should be identical.

```
open_system('rtwdemo_lct_start_term')
open_system('rtwdemo_lct_start_term/TestFault')
sim('rtwdemo_lct_start_term')
```



Integrate External C Functions That Pass Arguments as Multi-Dimensional Signals

This example shows how to use the Legacy Code Tool to integrate legacy C functions with multi-dimensional Signals.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C-MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a TLC block file and optional rtwmakecfg.m file that specifies how the generated code for a model calls the legacy code.

Provide the Legacy Function Specification

Legacy Code Tool functions take a specific data structure or array of structures as the argument. You can initialize the data structure by calling the function legacy_code() using 'initialize' as the first input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. For detailed help on the properties, call legacy_code('help'). The prototype of the legacy functions being called in this example is:

void array3d_add(real_T *y1, real_T *u1, real_T *u2, int32_T nbRows, int32_T nbCols, int32_T nbPages);

real_T is a typedef to double, and int32_T is a typedef to a 32-bit integer. The legacy source code is in the files ndarray_ops.h, and ndarray_ops.c.

```
% rtwdemo_sfun_ndarray_add
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_ndarray_add';
def.OutputFcnSpec = ['void array3d_add(double y1[size(u1,1)][size(u1,2)][size(u1,3)],
                     'double u1[][][], double u2[][][], ' ...
                    'int32 size(u1,1), int32 size(u1,2), int32 size(u1,3))'];
def.HeaderFiles   = {'ndarray_ops.h'};
def.SourceFiles   = {'ndarray_ops.c'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
```

y1 is a 3-D output signal of same dimensions as the 3-D input signal u1. Note that the last 3 arguments passed to the legacy function correspond to the number of element in each dimension of the 3-D input signal u1.

Generate an S-Function for Simulation

To generate a C-MEX S-function according to the description provided by the input argument 'def', call the function legacy_code() again with the first input set to 'sfcn_cmex_generate'. The S-function calls the legacy functions during simulation. The source code for the S-function is in the file rtwdemo_sfun_ndarray_add.c.

```
legacy_code('sfcn_cmex_generate', def);
```

Compile the Generated S-Function for Simulation

After you generate the C-MEX S-function source file, to compile the S-function for simulation with Simulink®, call the function legacy_code() again with the first input set to 'compile'.

```
legacy_code('compile', def);
```

```
### Start Compiling rtwdemo_sfun_ndarray_add
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc17a_538369_5
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_ndarray_add.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
```

```
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_ndarray_add
### Exit
```

Generate a TLC Block File for Code Generation

After you compile the S-function and use it in simulation, you can call the function legacy_code() again with the first input set to 'sfcn_tlc_generate' to generate a TLC block file. The block file specifies how the generated code for a model calls the legacy code. If you do not generate a TLC block file and you try to generate code for a model that includes the S-function, code generation fails. The TLC block file for the S-function is: rtwdemo_sfun_ndarray_add.tlc.

```
legacy_code('sfcn_tlc_generate', def);
```

Generate an rtwmakecfg.m File for Code Generation

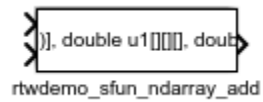
After you create the TLC block file, you can call the function legacy_code() again with the first input set to 'rtwmakecfg_generate' to generate an rtwmakecfg.m file to support code generation. If the required source and header files for the S-function are not in the same folder as the S-function, and you want to add these dependencies in the makefile produced during code generation, generate the rtwmakecfg.m file.

```
legacy_code('rtwmakecfg_generate', def);
```

Generate a Masked S-Function Block for Calling the Generated S-Function

After you compile the C-MEX S-function source, you can call the function legacy_code() again with the first input set to 'slblock_generate' to generate a masked S-function block that calls that S-function. The software places the block in a new model. From there you can copy it to an existing model.

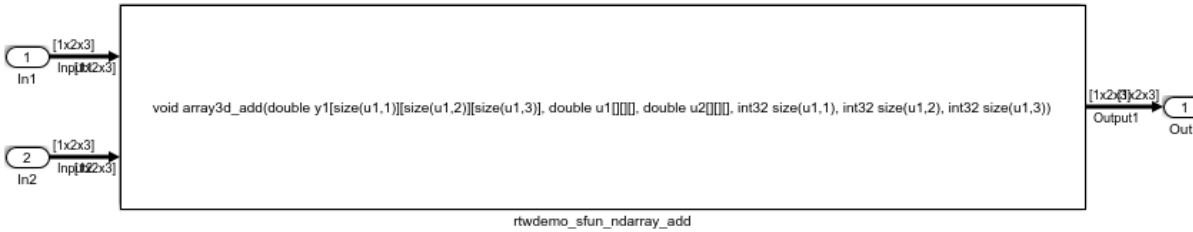
```
legacy_code('slblock_generate', def);
```



Showing the Generated Integration with Legacy Code

The model rtwdemo_lct_ndarray shows integration of the model with the legacy code. The subsystem ndarray_add serves as a harness for the call to the legacy C function.

```
open_system('rtwdemo_lct_ndarray')
open_system('rtwdemo_lct_ndarray/ndarray_add')
sim('rtwdemo_lct_ndarray')
```




```
This legacy function computes the addition of the 2 input signals:
   - Input1 and Input2 are dynamically sized 3D arrays
   - Output1 is a dynamically sized 3D array of same size as Input1
   - the last 3 function's arguments allow to pass the Input1's dimensions to the legacy function
```

Integrate External C Functions with a Block Sample Time Specified, Inherited, and Parameterized

This example shows how to use the Legacy Code Tool to integrate legacy C functions with the block's sample time specified, inherited and parameterized.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C-MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a TLC block file and optional rtwmakecfg.m file that specifies how the generated code for a model calls the legacy code.

Provide the Legacy Function Specification

Legacy Code Tool functions take a specific data structure or array of structures as the argument. You can initialize the data structure by calling the function legacy_code() using 'initialize' as the first input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. For detailed help on the properties, call legacy_code('help'). The prototype of the legacy functions being called in this example is:

FLT gainScalar(const FLT in, const FLT gain)

FLT is a typedef to float. The legacy source code is in the files your_types.h, gain.h, and gainScalar.c.

```
defs = [];

% rtwdemo_sfun_st_inherited
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_st_inherited';
def.OutputFcnSpec = 'single y1 = gainScalar(single u1, single p1)';
def.HeaderFiles   = {'gain.h'};
def.SourceFiles   = {'gainScalar.c'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
defs = [defs; def];

% rtwdemo_sfun_st_fixed
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_st_fixed';
def.OutputFcnSpec = 'single y1 = gainScalar(single u1, single p1)';
def.HeaderFiles   = {'gain.h'};
def.SourceFiles   = {'gainScalar.c'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
def.SampleTime    = [2 1];
defs = [defs; def];

% rtwdemo_sfun_st_parameterized
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_st_parameterized';
def.OutputFcnSpec = 'single y1 = gainScalar(single u1, single p1)';
def.HeaderFiles   = {'gain.h'};
def.SourceFiles   = {'gainScalar.c'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
def.SampleTime    = 'parameterized';
defs = [defs; def];
```

Generate S-Functions for Simulation

To generate C-MEX S-functions according to the description provided by the input argument 'defs', call the function legacy_code() again with the first input set to 'sfcn_cmex_generate'. The S-functions call the legacy functions during simulation. The source code for the S-functions is in the files rtwdemo_sfun_st_inherited.c and rtwdemo_sfun_st_fixed.c. rtwdemo_sfun_st_parameterized.c.

```
legacy_code('sfcn_cmex_generate', defs);
```

Compile the Generated S-Functions for Simulation

After you generate the C-MEX S-function source files, to compile the S-functions for simulation with Simulink®, call the function legacy_code() again with the first input set to 'compile'.

```
legacy_code('compile', defs);
```

```
### Start Compiling rtwdemo_sfun_st_inherited
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc17a_538369_5
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_st_inherited.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_sr
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_st_inherited
### Exit

### Start Compiling rtwdemo_sfun_st_fixed
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc17a_538369_5
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_st_fixed.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src',
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_st_fixed
### Exit

### Start Compiling rtwdemo_sfun_st_parameterized
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc17a_538369_5
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_st_parameterized.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lc
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_st_parameterized
### Exit
```

Generate TLC Block Files for Code Generation

After you compile the S-functions and use them in simulation, you can call the function legacy_code() again with the first input set to 'sfcn_tlc_generate' to generate

TLC block files. Block files specify how the generated code for a model calls the legacy code. If you do not generate TLC block files and you try to generate code for a model that includes the S-functions, code generation fails. The TLC block files for the S-functions are rtwdemo_sfun_st_inherited.tlc and rtwdemo_sfun_st_fixed.tlc. rtwdemo_sfun_st_parameterized.tlc.

```
legacy_code('sfcn_tlc_generate', defs);
```

Generate an rtwmakecfg.m File for Code Generation

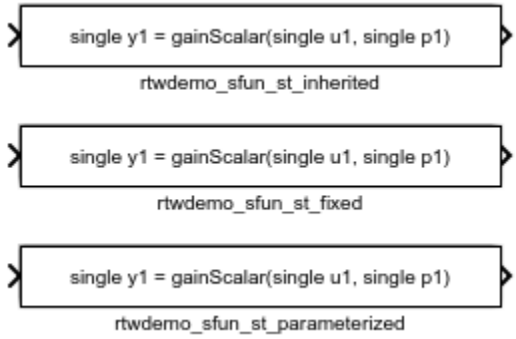
After you create the TLC block files, you can call the function legacy_code() again with the first input set to 'rtwmakecfg_generate' to generate an rtwmakecfg.m file to support code generation. If the required source and header files for the S-functions are not in the same folder as the S-functions, and you want to add these dependencies in the makefile produced during code generation, generate the rtwmakecfg.m file.

```
legacy_code('rtwmakecfg_generate', defs);
```

Generate Masked S-Function Blocks for Calling the Generated S-Functions

After you compile the C-MEX S-function source, you can call the function legacy_code() again with the first input set to 'slblock_generate' to generate masked S-function blocks that call the S-functions. The software places the blocks in a new model. From there you can copy them to an existing model.

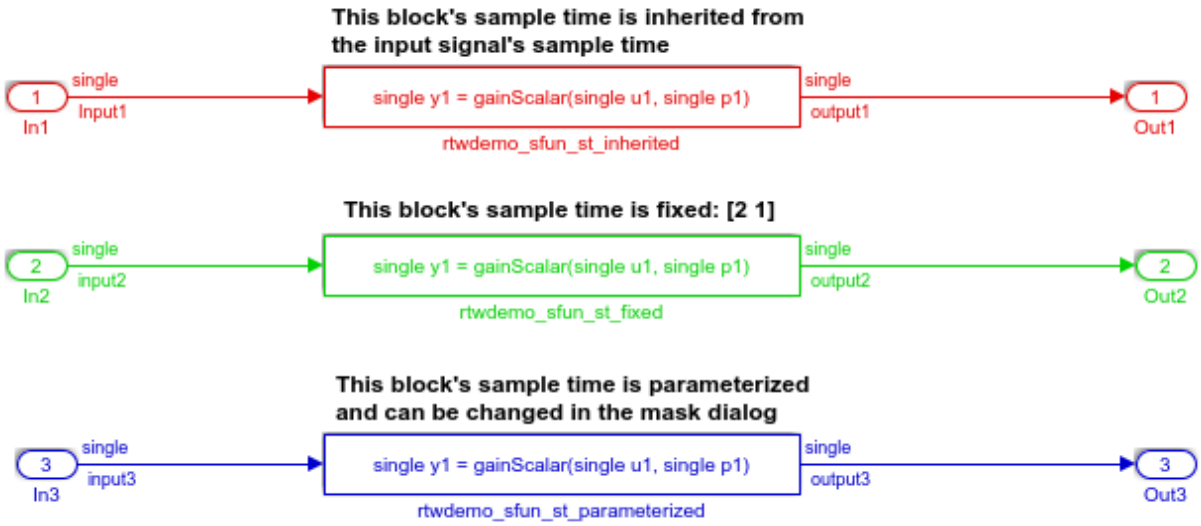
```
legacy_code('slblock_generate', defs);
```



Show the Generated Integration with Legacy Code

The model rtwdemo_lct_sampletime shows integration of the model with the legacy code. The subsystem sample_time serves as a harness for the calls to the legacy C functions, with unit delays serving to store the previous output values.

```
open_system('rtwdemo_lct_sampletime')
open_system('rtwdemo_lct_sampletime/sample_time')
sim('rtwdemo_lct_sampletime')
```



See Also

`legacy_code`

Related Examples

- "Integrate C Functions Using Legacy Code Tool" (Simulink)
- "Call External C Code from Model and Generated Code"

External Code Integration Examples

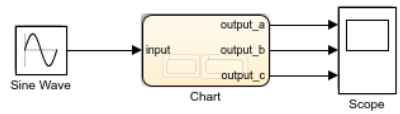
This topic shows various scenarios of external code integration.

Insert External C and C++ Code Into Stateflow Charts for Code Generation

This example shows how to use Stateflow® to integrate external code into a model.

Open Model

```
model='rtwdemo_sfcustom';
open_system(model);
```



Integrating c-code into model

Custom written c-files can be easily integrated into Stateflow models. This code can be used to augment Stateflow's capabilities or to take advantage of legacy code.

To add custom c-files open the simulation target and enter the following:

Include Code - Header that defines functions, structures, and data to be accessible by Stateflow.
Include Path - Path to this include file.
Source Files - c-files that define the functions and data accessible by Stateflow.

Open simulation custom code settings

If generating code via Simulink Coder, these same settings must also be added to the Model's configuration parameters custom code settings.

Open Code Generation custom code settings

Build model using Simulink Coder

Calling c-code from Stateflow

Functions
Custom c-code functions can be called from Stateflow using the same syntax as graphical function calls. The statement takes the form:

result = my_custom_function(in_args);

Structures
Variables of structure type can be accessed in Stateflow via the "dot" notation. The expression takes the form:

result = my_var.my_field;

To view the custom source for this examplenstration double click the links below.

Open my_header.h

Open my_function.c

Calling C++ code from Stateflow

Custom C++ code can also be integrated into Stateflow and Simulink Coder. Double-click below for a example.

Open rtwdemo_sfcpp

Additional documentation

Additional documentation is available for integrating C and C++ code in your modelby double-clicking the link below.

C-Code integration

C++-Code integration

Integrate Code

1. The example includes the custom header file `my_header.c` and the custom source file `my_function.c`.

```
%Open files my_header.h and my_function.c
eval('edit my_header.h')
eval('edit my_function.c')
```

2. On the Configuration Parameters dialog box **Simulation Target** pane, enter the custom source file and header file. Also enter additional include directories and source files.

In this example, the custom header file `my_header.c` and source file `my_function.c` are entered on the **Simulation Target** pane.

```
%Open Configuration Parameters dialog box
slCfgPrmDlg(model,'Open');
slCfgPrmDlg(bdroot,'TurnToPage','Simulation Target');
```

3. If you generate code with Simulink Coder®, on the Configuration Parameters dialog box **Code Generation > Custom Code** pane, enter the same custom source file and header file. Also enter the same additional include directories and source files.

In this example, the custom header file `my_header.c` and source file `my_function.c` are entered on the **Code Generation > Custom Code** pane.

```
%Open Configuration Parameters dialog box
slCfgPrmDlg(model,'Open');
slCfgPrmDlg(bdroot,'TurnToPage','Code Generation/Custom Code');
```

Generate Code

```
rtwbuild('rtwdemo_sfcustom')
```

```
### Starting build procedure for model: rtwdemo_sfcustom
### Successful completion of build procedure for model: rtwdemo_sfcustom
```

Call C Code from Stateflow

To call custom C code functions from Stateflow, use the same syntax as graphical function calls: `result = my_custom_function(in_args);`

To call variables of structure type, use the dot notation: `result = my_var.my_field;`

See Also

- Include Custom C Code in Simulation Targets for Library Models
- Integrate Custom C++ Code for Simulation

Close Model

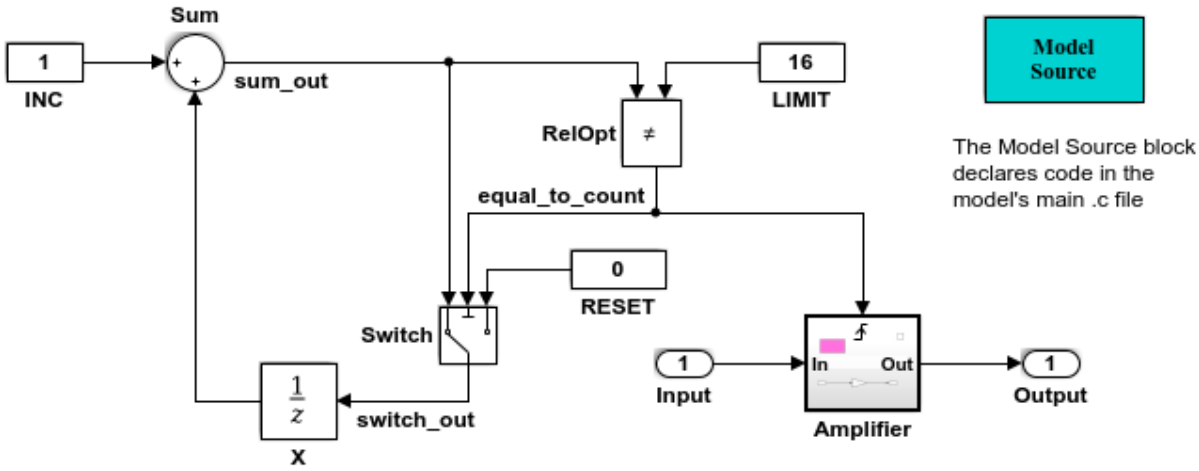
```
rtwdemoclean;
close_system('rtwdemo_sfcustom',0);
```

Integrate External C Code Into Generated Code By Using Custom Code Blocks and Model Configuration Parameters

This example shows how to place external code in generated code by using custom code blocks and model configuration parameters.

1. Open the model `rtwdemo_slcustcode`.

```
open_system('rtwdemo_slcustcode')
```



2. Open the Model Configuration Parameters dialog box and navigate to the **Custom Code** pane.

3. Examine the settings for parameters **Source file** and **Initialize function**.

- **Source file** specifies a comment and sets the variable `GLOBAL_INT2` to -1.
- **Initialize function** intializes the variable `GLOBAL_INT2` to 1.

4. Close the dialog box.

5. Double-click the Model Source block. The **Top of Model Source** field specifies that the code generator declare the variable `GLOBAL_INT1` and set it to 0 at the top of the generated file `rtwdemo_slcustcode.c`.

6. Open the triggered subsystem `Amplifier`. The subsystem includes the System Outputs block. The code generator places code that you specify in that block in the generated code for the nearest parent atomic subsystem. In this case, the code generator places the external code in the generated code for the `Amplifier` subsystem. The external code:

- Declares the pointer variable `*intPtr` and intitializes it with the value of variable `GLOBAL_INT1`.
- Sets the pointer variable to -1 during execution.
- Resets the pointer variable to 0 before exiting.

7. Generate code and a code generation report.

8. Examine the code in the generated source file `rtwdemo_slcustcode.c`. At the top of the file, after the `#include` statements, you find the following declaration code. The example specifies the first declaration with the **Source file** configuration parameter and the second declaration with the Model Source block.

```
int_T GLOBAL_INT2 = -1;

int_T GLOBAL_INT1 = 0;
```

The Output function for the `Amplifier` subsystem includes the following code, which shows the external code integrated with generated code that applies the gain. The example specifies the three lines of code for the pointer variable with the System Output block in the `Amplifier` subsystem.

```
int_T *intPtr = &GLOBAL_INT1;

*intPtr = -1;
```

```
rtwdemo_slcustcode_Y.Output = rtwdemo_slcustcode_U.Input << 1;

*intPtr = 0;
```

The following assignment appears in the model initialize entry-point function. The example specifies this assignment with the **Initialize function** configuration parameter.

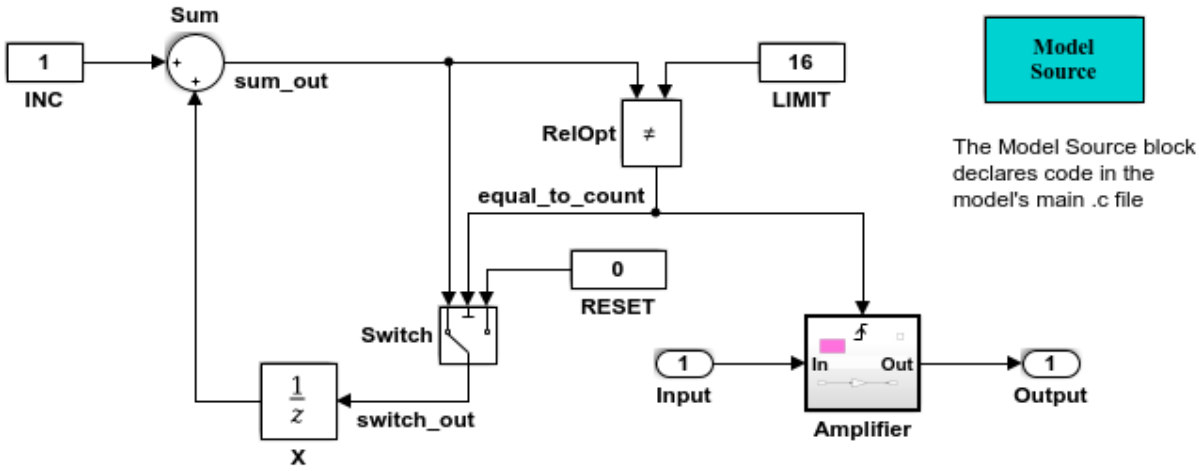
```
GLOBAL_INT2 = 1;
```

Integrate External C Code Into Generated Code By Using Custom Code Blocks and Model Configuration Parameters

This example shows how to place external code in generated code by using custom code blocks and model configuration parameters.

1. Open the model `rtwdemo_slcustcode`.

```
open_system('rtwdemo_slcustcode')
```



2. Open the Model Configuration Parameters dialog box and navigate to the **Custom Code** pane.

3. Examine the settings for parameters **Source file** and **Initialize function**.

- **Source file** specifies a comment and sets the variable `GLOBAL_INT2` to -1.
- **Initialize function** intializes the variable `GLOBAL_INT2` to 1.

4. Close the dialog box.

5. Double-click the Model Source block. The **Top of Model Source** field specifies that the code generator declare the variable `GLOBAL_INT1` and set it to 0 at the top of the generated file `rtwdemo_slcustcode.c`.

6. Open the triggered subsystem `Amplifier`. The subsystem includes the System Outputs block. The code generator places code that you specify in that block in the generated code for the nearest parent atomic subsystem. In this case, the code generator places the external code in the generated code for the `Amplifier` subsystem. The external code:

- Declares the pointer variable `*intPtr` and intitializes it with the value of variable `GLOBAL_INT1`.
- Sets the pointer variable to -1 during execution.
- Resets the pointer variable to 0 before exiting.

7. Generate code and a code generation report.

8. Examine the code in the generated source file `rtwdemo_slcustcode.c`. At the top of the file, after the `#include` statements, you find the following declaration code. The example specifies the first declaration with the **Source file** configuration parameter and the second declaration with the Model Source block.

```
int_T GLOBAL_INT2 = -1;

int_T GLOBAL_INT1 = 0;
```

The Output function for the `Amplifier` subsystem includes the following code, which shows the external code integrated with generated code that applies the gain. The example specifies the three lines of code for the pointer variable with the System Output block in the `Amplifier` subsystem.

```
int_T *intPtr = &GLOBAL_INT1;

*intPtr = -1;
```

```
rtwdemo_slcustcode_Y.Output = rtwdemo_slcustcode_U.Input << 1;

*intPtr = 0;
```

The following assignment appears in the model initialize entry-point function. The example specifies this assignment with the **Initialize function** configuration parameter.

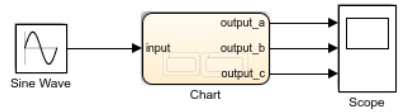
```
GLOBAL_INT2 = 1;
```

Insert External C and C++ Code Into Stateflow Charts for Code Generation

This example shows how to use Stateflow® to integrate external code into a model.

Open Model

```
model='rtwdemo_sfcustom';
open_system(model);
```



Integrating c-code into model

Custom written c-files can be easily integrated into Stateflow models. This code can be used to augment Stateflow's capabilities or to take advantage of legacy code.

To add custom c-files open the simulation target and enter the following:

Include Code - Header that defines functions, structures, and data to be accessible by Stateflow.
Include Path - Path to this include file.
Source Files - c-files that define the functions and data accessible by Stateflow.

Open simulation custom code settings

If generating code via Simulink Coder, these same settings must also be added to the Model's configuration parameters custom code settings.

Open Code Generation custom code settings

Build model using Simulink Coder

Calling c-code from Stateflow

Functions
Custom c-code functions can be called from Stateflow using the same syntax as graphical function calls. The statement takes the form:

result = my_custom_function(in_args);

Structures
Variables of structure type can be accessed in Stateflow via the "dot" notation. The expression takes the form:

result = my_var.my_field;

To view the custom source for this examplenstration double click the links below.

Open my_header.h

Open my_function.c

Calling C++ code from Stateflow

Custom C++ code can also be integrated into Stateflow and Simulink Coder. Double-click below for a example.

Open rtwdemo_sfcpp

Additional documentation

Additional documentation is available for integrating C and C++ code in your modelby double-clicking the link below.

C-Code integration

C++-Code integration

Integrate Code

1. The example includes the custom header file `my_header.c` and the custom source file `my_function.c`.

```
%Open files my_header.h and my_function.c
eval('edit my_header.h')
eval('edit my_function.c')
```

2. On the Configuration Parameters dialog box **Simulation Target** pane, enter the custom source file and header file. Also enter additional include directories and source files.

In this example, the custom header file `my_header.c` and source file `my_function.c` are entered on the **Simulation Target** pane.

```
%Open Configuration Parameters dialog box
slCfgPrmDlg(model,'Open');
slCfgPrmDlg(bdroot,'TurnToPage','Simulation Target');
```

3. If you generate code with Simulink Coder®, on the Configuration Parameters dialog box **Code Generation > Custom Code** pane, enter the same custom source file and header file. Also enter the same additional include directories and source files.

In this example, the custom header file `my_header.c` and source file `my_function.c` are entered on the **Code Generation > Custom Code** pane.

```
%Open Configuration Parameters dialog box
slCfgPrmDlg(model,'Open');
slCfgPrmDlg(bdroot,'TurnToPage','Code Generation/Custom Code');
```

Generate Code

```
rtwbuild('rtwdemo_sfcustom')
```

```
### Starting build procedure for model: rtwdemo_sfcustom
### Successful completion of build procedure for model: rtwdemo_sfcustom
```

Call C Code from Stateflow

To call custom C code functions from Stateflow, use the same syntax as graphical function calls: `result = my_custom_function(in_args);`

To call variables of structure type, use the dot notation: `result = my_var.my_field;`

See Also

- Include Custom C Code in Simulation Targets for Library Models
- Integrate Custom C++ Code for Simulation

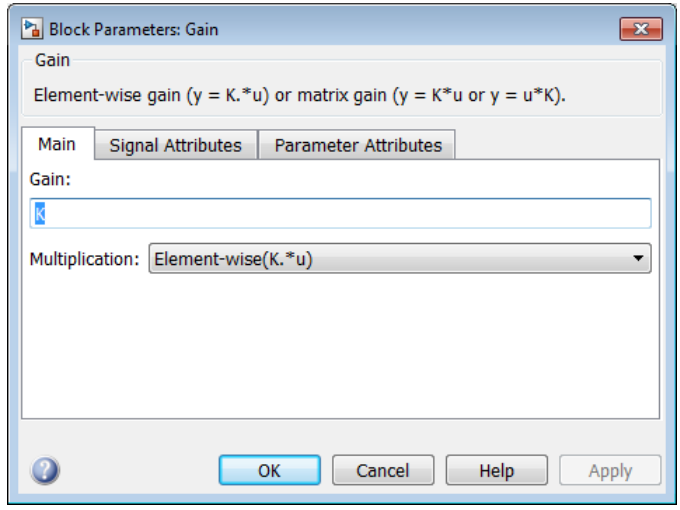
Close Model

```
rtwdemoclean;
close_system('rtwdemo_sfcustom',0);
```

Automate S-Function Generation with S-Function Builder

The **Generate S-function** feature automates the process of generating an S-function from a subsystem. In addition, the **Generate S-function** feature presents a display of parameters used within the subsystem, and lets you declare selected parameters tunable.

As an example, consider `SourceSubsys`, the same subsystem illustrated in the example "Create S-Function Blocks from a Subsystem" on page 46-37. The objective is to automatically extract `SourceSubsys` from the model and build an S-Function block from it, as in the previous example. In addition, the workspace variable `K` , which is the gain factor of the Gain block within `SourceSubsys` (as shown in the Gain block parameter dialog box below), is declared and generated as a tunable variable.



To auto-generate an S-function from `SourceSubsys` with tunable parameter `K`,

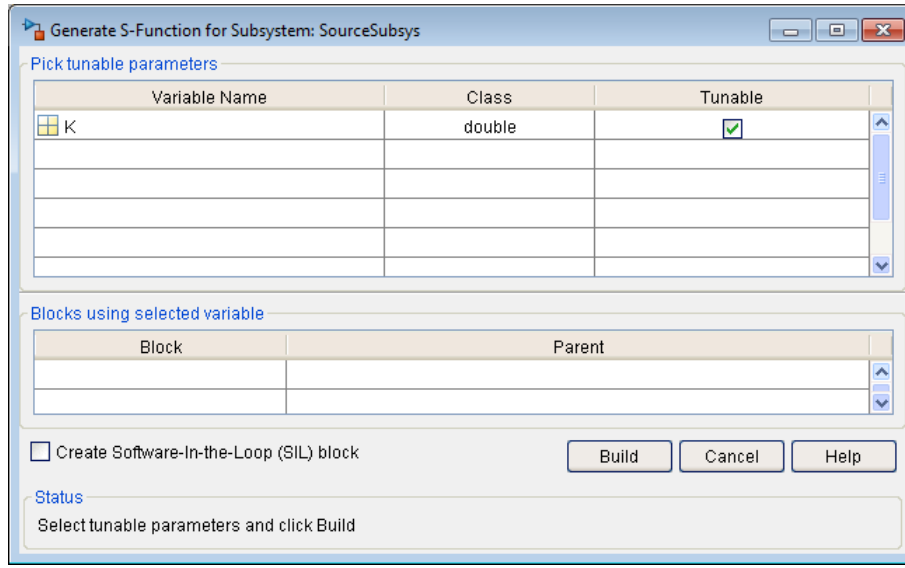
1. With the `SourceSubsys` model open, click the subsystem to select it.
2. From the **Code** menu, select **C/C++ Code** > **Generate S-Function**. This menu item is enabled when a subsystem is selected in the current model.

   Alternatively, you can right-click the subsystem and select **C/C++ Code** > **Generate S-Function** from the subsystem block's context menu.
3. The **Generate S-Function** window is displayed (see the next figure). This window shows variables (or data objects) that are referenced as block parameters in the subsystem, and lets you declare them as tunable.

The upper pane of the window displays three columns:

- **Variable Name**: name of the parameter.
- **Class**: If the parameter is a workspace variable, its data type is shown. If the parameter is a data object, its name and class is shown
- **Tunable**: Lets you select tunable parameters. To declare a parameter tunable, select the check box. In the next figure, the parameter K is declared tunable.

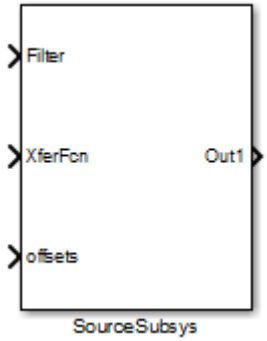
When you select a parameter in the upper pane, the lower pane shows the blocks that reference the parameter, and the parent system of each such block.



Generate S-Function Window

4 After selecting tunable parameters, click the **Build** button. This initiates code generation and compilation of the S-function, using the S-function target. The **Create New Model** option is automatically enabled.

5 The build process displays status messages in the MATLAB Command Window. When the build completes, the tunable parameters window closes, and a new untitled model window opens.



6 The model window contains an S-Function block with the same name as the subsystem from which the block was generated (in this example, `SourceSubsys`). Optionally, you can save the generated model containing the generated block.

7 The generated code for the S-Function block is stored in the current working folder. The following files are written to the top level folder:

- *`subsys`*`_sf.c` or `.cpp`, where *`subsys`* is the subsystem name (for example, `SourceSubsys_sf.c`)
- *`subsys`*`_sf.h`
- *`subsys`*`_sf.`*`mexext`*, where *`mexext`* is a platform-dependent MEX-file extension (for example, `SourceSubsys_sf.mexw64`)

The source code for the S-function is written to the subfolder *`subsys`*`_sfcn_rtw`. The top-level `.c` or `.cpp` file is a stub file that simply contains an include directive that you can use to interface other C/C++ code to the generated code.

Note: For a list of files required to deploy your S-Function block for simulation or code generation, see "Required Files for S-Function Deployment" on page 46-35.

---

8 The generated S-Function block has inports and outports whose widths and sample times correspond to those of the original model.

The following code, from the `mdlOutputs` routine of the generated S-function code (in `SourceSubsys_sf.c`), shows how the tunable variable `K` is referenced by using calls to the MEX API.

```
static void mdlOutputs(SimStruct *S, int_T tid)
```

```
...

/* Gain: '<S1>/Gain' incorporates:
     *  Sum: '<S1>/Sum'
     */
    rtb_Gain_n[0] = (rtb_Product_p + (*(((const
      real_T**)ssGetInputPortSignalPtrs(S, 2))[0]))) * (*(real_T
      *)(mxGetData(K(S))));
    rtb_Gain_n[1] = (rtb_Product_p + (*(((const
      real_T**)ssGetInputPortSignalPtrs(S, 2))[1]))) * (*(real_T
      *)(mxGetData(K(S))));
```

- In automatic S-function generation, the **Use Value for Tunable Parameters** option is cleared or at the command line is set to `'off'`.
- Use a MEX S-function wrapper only in the MATLAB version in which the wrapper is created.

If you specify paths and files with absolute

Macro Parameters

Suppose that you apply a custom storage class such as `Define` to a `Simulink.Parameter` object so that the parameter appears as a macro in the generated code. If you use the parameter object inside a subsystem from which you generate an ERT S-function, you cannot select the parameter object as a tunable parameter. Instead, the S-function code generator applies the custom storage class to the parameter object. This generation of macros in the S-function code allows you to generate S-functions from subsystems that contain variant elements, such as Variant Subsystem blocks, that you configure to produce preprocessor conditionals in the generated code. However, you cannot change the value of the parameter during simulation of the S-function.

To select the parameter object as a tunable parameter, apply a different storage class or custom storage class. Custom storage classes that treat parameters as macros include `Define`, `ImportedDefine`, `CompilerFlag`, and custom storage classes that you create by setting **Data initialization** to `Macro` in the Custom Storage Class Designer. If you use a non-macro storage class or custom storage class, you cannot use the parameter object as a variant control variable and generate preprocessor conditionals.

If you apply a custom storage class that treats the parameter object as an imported macro, provide the macro definition before you generate the ERT S-function. For example, suppose you apply the custom storage class `ImportedDefine` to a `Simulink.Parameter` object, and use the parameter object as a variant control variable in the subsystem. If you set the custom attribute `HeaderFile` to `'myHdr.h'`, when you

generate the S-function, place the custom header file `myHdr.h` in the current folder. The generated S-function uses the macro value from your header file instead of the value from the `Value` property of the parameter object.

To use a macro that you define through a compiler option, for example by applying the custom storage class `CompilerFlag`, use the model configuration parameter **Configuration Parameters** > **Code Generation** > **Custom Code** > **Additional build information** > **Defines** to specify the compiler option. For more information, see Code Generation Pane: Custom Code: Additional Build Information: Defines (Simulink Coder).

See Also

`legacy_code`

More About

- "S-Functions and Code Generation" on page 11-2
- "Import Calls to External Code into Generated Code with Legacy Code Tool" on page 11-7

Write S-Function and TLC Files By Hand

You can choose from several approaches for writing S-function and TLC files by hand.

Write Noninlined S-Function and TLC Files

About Noninlined S-Functions

Noninlined S-functions are identified by the *absence* of an `sfunction.tlc` file for your S-function. The filename varies depending on your platform. For example, on a 64–bit Microsoft Windows system, the file name would be `sfunction.mexw64`. Type mexext in the MATLAB Command Window to see which extension your system uses.

Guidelines for Writing Noninlined S-Functions

- The MEX-file cannot call MATLAB functions.
- If the MEX-file uses functions in the MATLAB External Interface libraries, include the header file `cg_sfun.h` instead of `mex.h` or `simulink.c`. To handle this case, include the following lines at the end of your S-function:

```
#ifdef MATLAB_MEX_FILE /* Is this file being compiled as a MEX-file? */
#include "simulink.c"  /* MEX-file interface mechanism */
#else
#include "cg_sfun.h"   /* Code generation registration function */
#endif
```

- Use only MATLAB API function that the code generator supports, which include:

```
mxGetEps
mxGetInf
mxGetM
mxGetN
mxGetNaN
mxGetPr
mxGetScalar
mxGetString
mxIsEmpty
mxIsFinite
mxIsInf
```

- MEX library calls are not supported in generated code. To use such calls in MEX-file and not in the generated code, conditionalize the code as follows:

  ```
  #ifdef MATLAB_MEX_FILE
  #endif
  ```

- Use only full matrices that contain only real data.
- Do not specify a return value for calls to `mxGetString` . If you do specify a return value, the MEX-file will not compile. Instead, use the function's second input argument, which returns a pointer to a character vector.
- Make sure that the `#define` *`s-function_name`* statement is correct. The S-function name that you specify must match the S-function's filename.
- Use the data types `real_T` and `int_T` instead of `double` and `int`, if possible. The data types `real_T` and `int_T` are more generic and can be used in multiple environments.
- Provide the build process with the names of the modules used to build the S-function. You can do this by using a template make file, the `set_param` function, or the `S-function modules` field of the S-Function block parameters dialog box. For example, suppose you build your S-function with the following command:

  ```
  mex sfun_main.c sfun_module1.c sfun_module2.c
  ```

  You can then use the following call to `set_param` to include the required modules:

  ```
  set_param(sfun_block, "SFunctionModules","sfun_module1 sfun_module2")
  ```

  When you are ready to generate code, force the code generator to rebuild the top model, as explained in “Control Regeneration of Top Model Code” (Simulink Coder).

Noninlined S-Function Parameter Type Limitations

Parameters to noninlined S-functions can be of the following types only:

- Double precision
- Characters in scalars, vectors, or 2-D matrices

For more flexibility in the type of parameters you can supply to S-functions or the operations in the S-function, inline your S-function and consider using an `mdlRTW` S-function routine.

Use of other functions from the MATLAB `matrix.h` API or other MATLAB APIs, such as `mex.h` and `mat.h`, is not supported. If you call unsupported APIs from an S-function

source file, compiler errors occur. See the files *matlabroot*/rtw/c/src/rt_matrx.h and *matlabroot*/rtw/c/src/rt_matrx.c for details on supported MATLAB API functions.

If you use mxGetPr on an empty matrix, the function does not return NULL; rather, it returns a random value. Therefore, you should protect calls to mxGetPr with mxIsEmpty.

Write Wrapper S-Function and TLC Files

This topic describes how to create S-functions that work seamlessly with the Simulink and code generator products using the *wrapper* concept. This topic begins by describing how to interface your algorithms in Simulink models by writing MEX S-function wrappers (*sfunction*.mex). It finishes with a description of how to direct the code generator to insert your algorithm into the generated code by creating a TLC S-function wrapper (*sfunction*.tlc).

MEX S-Function Wrapper

Creating S-functions using an S-function wrapper allows you to insert C/C++ code algorithms in Simulink models and the generated code with little or no change to your original C/C++ function. A *MEX S-function wrapper* is an S-function that calls code that resides in another module. A *TLC S-function wrapper* is a TLC file that specifies how the code generator should call your code (the same code that was called from the C MEX S-function wrapper).

Note: A MEX S-function wrapper must only be used in the MATLAB version in which the wrapper is created.

Suppose you have an algorithm (that is, a C function) called my_alg that resides in the file my_alg.c. You can integrate my_alg into a Simulink model by creating a MEX S-

function wrapper (for example, `wrapsfcn.c`). Once this is done, a Simulink model can call `my_alg` from an S-Function block. However, the Simulink S-function contains a set of empty functions that the Simulink engine requires for various API-related purposes. For example, although only `mdlOutputs` calls `my_alg`, the engine calls `mdlTerminate` as well, even though this S-function routine performs no action.

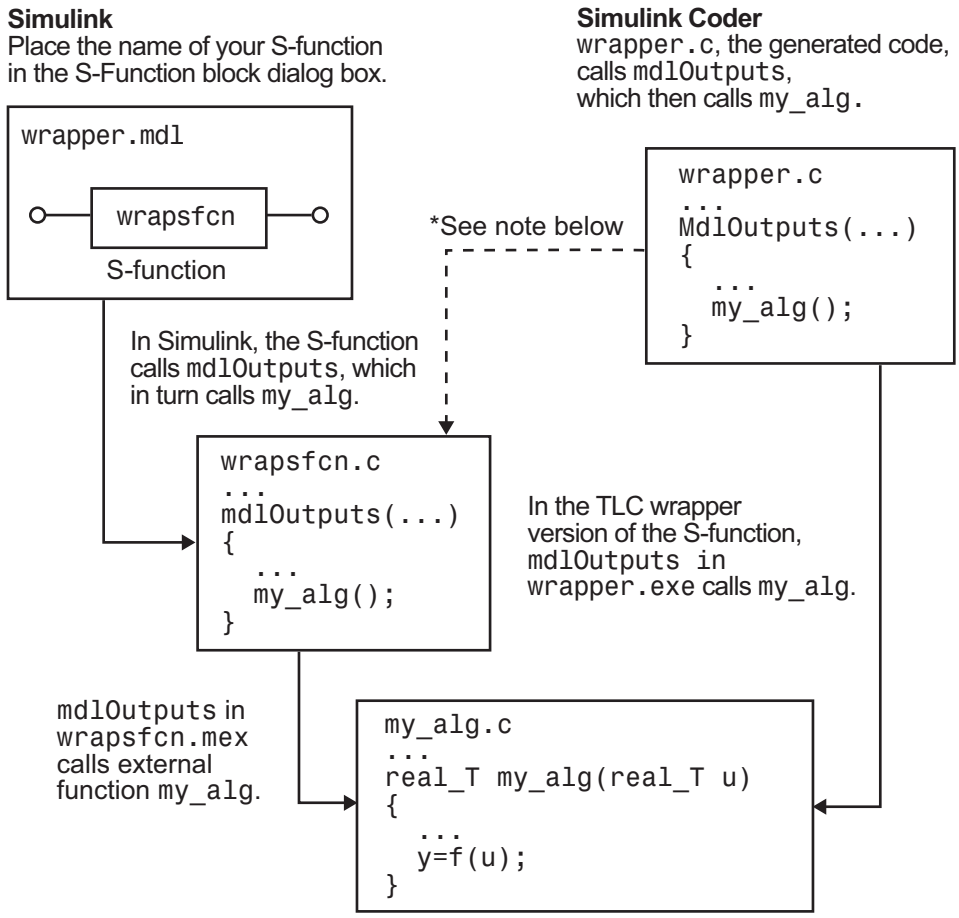
You can integrate `my_alg` into generated code (that is, embed the call to `my_alg` in the generated code) by creating a TLC S-function wrapper (for example, `wrapsfcn.tlc`). The advantage of creating a TLC S-function wrapper is that the empty function calls can be eliminated and the overhead of executing the `mdlOutputs` function and then the `my_alg` function can be eliminated.

Wrapper S-functions are useful when you are creating new algorithms that are procedural in nature or when you are integrating legacy code into a Simulink model. However, if you want to create code that is

- Interpretive in nature (that is, highly parameterized by operating modes)
- Heavily optimized (that is, no extra tests to decide what mode the code is operating in)

then you must create a *fully inlined TLC file* for your S-function.

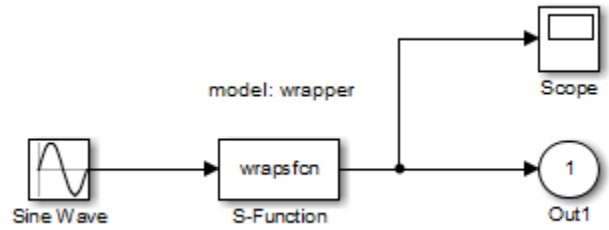
The next figure shows the wrapper S-function concept.



*The dotted line is the path taken if the S-function does not have a TLC wrapper file. If there is no TLC wrapper file, the generated code calls `mdlOutputs`.

Using an S-function wrapper to import algorithms in your Simulink model means that the S-function serves as an interface that calls your C/C++ algorithms from `mdlOutputs`. S-function wrappers have the advantage that you can quickly integrate large standalone C /C++ programs into your model without having to make changes to the code.

The following sample model includes an S-function wrapper.



There are two files associated with the `wrapsfcn` block, the S-function wrapper and the C/C++ code that contains the algorithm. The S-function wrapper code for `wrapsfcn.c` appears below. The first three statements do the following:

1. Defines the name of the S-function (what you enter in the Simulink S-Function block dialog).
2. Specifies that the S-function is using the level 2 format.
3. Provides access to the `SimStruct` data structure, which contains pointers to data used during simulation and code generation and defines macros that store data in and retrieve data from the `SimStruct`.

For more information, see "Templates for C S-Functions" (Simulink).

```
#define S_FUNCTION_NAME wrapsfcn
#define S_FUNCTION_LEVEL 2
#include "simstruc.h"

extern real_T my_alg(real_T u);  /* Declare my_alg as extern */

/*
 * mdlInitializeSizes - initialize the sizes array
 */
static void mdlInitializeSizes(SimStruct *S)
{

    ssSetNumSFcnParams( S, 0); /*number of input arguments*/

    if (!ssSetNumInputPorts(S, 1)) return;
    ssSetInputPortWidth(S, 0, 1);
    ssSetInputPortDirectFeedThrough(S, 0, 1);

    if (!ssSetNumOutputPorts(S,1)) return;
    ssSetOutputPortWidth(S, 0, 1);

    ssSetNumSampleTimes( S, 1);
}

/*
```

```
 * mdlInitializeSampleTimes - indicate that this S-function runs
 * at the rate of the source (driving block)
 */
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
}


/*
 * mdlOutputs - compute the outputs by calling my_alg, which
 * resides in another module, my_alg.c
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
    InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);
    real_T            *y    = ssGetOutputPortRealSignal(S,0);
    *y = my_alg(*uPtrs[0]); /* Call my_alg in mdlOutputs */
 }
/*
 * mdlTerminate - called when the simulation is terminated.
 */
static void mdlTerminate(SimStruct *S)
{
}

#ifdef MATLAB_MEX_FILE /* Is this file being compiled as a MEX-file? */
#include "simulink.c" /* MEX-file interface mechanism */
#else
#include "cg_sfun.h" /* Code generation registration function */
#endif
```

The S-function routine `mdlOutputs` contains a function call to `my_alg`, which is the C function containing the algorithm that the S-function performs. This is the code for `my_alg.c`:

```
#ifdef MATLAB_MEX_FILE
#include "tmwtypes.h"
#else
#include "rtwtypes.h"
#endif
real_T my_alg(real_T u)
{
return(u * 2.0);
}
```

For more information, see “Manage Build Process File Dependencies” (Simulink Coder).

The wrapper S-function `wrapsfcn` calls `my_alg`, which computes `u * 2.0`. To build `wrapsfcn.mex`, use the following command:

```
mex wrapsfcn.c my_alg.c
```

TLC S-Function Wrapper

This topic describes how to inline the call to `my_alg` in the `mdlOutputs` section of the generated code. In the above example, the call to `my_alg` is embedded in the `mdlOutputs` section as

```
*y = my_alg(*uPtrs[0]);
```

When you are creating a TLC S-function wrapper, the goal is to embed the same type of call in the generated code.

It is instructive to look at how the code generator executes S-functions that are not inlined. A noninlined S-function is identified by the absence of the file *`sfunction`*`.tlc` and the existence of *`sfunction`*`.mex`. When generating code for a noninlined S-function, the code generator produces a call to `mdlOutputs` through a function pointer that, in this example, then calls `my_alg`.

The wrapper example contains one S-function, `wrapsfcn.mex`. You must compile and link an additional module, `my_alg`, with the generated code. To do this, specify

```
set_param('wrapper/S-Function','SFunctionModules','my_alg')
```

Code Overhead for Noninlined S-Functions

The code generated when using `grt.tlc` as the system target file *without* `wrapsfcn.tlc` is

```
<Generated code comments for wrapper model with noninlined wrapsfcn S-function>

#include <math.h>
#include <string.h>
#include "wrapper.h"
#include "wrapper.prm"

/* Start the model */
void mdlStart(void)
{
  /* (start code not required) */
}

/* Compute block outputs */
void mdlOutputs(int_T tid)
{
  /* Sin Block: <Root>/Sin */
  rtB.Sin = rtP.Sin.Amplitude *
    sin(rtP.Sin.Frequency * ssGetT(rtS) + rtP.Sin.Phase);

  /* Level2 S-Function Block: <Root>/S-Function (wrapsfcn) */
```

```
  {
    /* Noninlined S-functions create a SimStruct object and
     * generate a call to S-function routine mdlOutputs
     */
    SimStruct *rts = ssGetSFunction(rtS, 0);
    sfcnOutputs(rts, tid);
  }

  /* Outport Block: <Root>/Out */
  rtY.Out = rtB.S_Function;
}

/* Perform model update */
void mdlUpdate(int_T tid)
{
  /* (update code not required) */
}

/* Terminate function */
void mdlTerminate(void)
{
  /* Level2 S-Function Block: <Root>/S-Function (wrapsfcn) */
  {
/* Noninlined S-functions require a SimStruct object and
     * the call to S-function routine mdlTerminate
     */
    SimStruct *rts = ssGetSFunction(rtS, 0);
    sfcnTerminate(rts);
  }
}

#include "wrapper.reg"

/* [EOF] wrapper.c */
```

In addition to the overhead outlined above, the `wrapper.reg` generated file contains the initialization of the `SimStruct` for the wrapper S-Function block. There is one child `SimStruct` for each S-Function block in your model. You can significantly reduce this overhead by creating a TLC wrapper for the S-function.

How to Inline

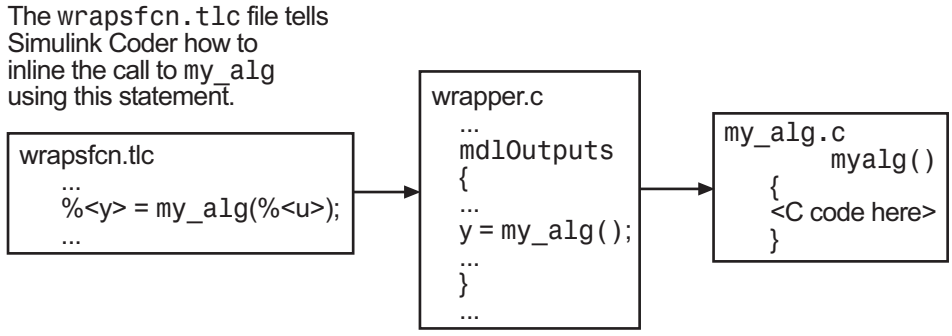
The generated code makes the call to your S-function, `wrapsfcn.c`, in `mdlOutputs` by using this code:

```
SimStruct *rts = ssGetSFunction(rtS, 0);
sfcnOutputs(rts, tid);
```

This call has computational overhead associated with it. First, the Simulink engine creates a `SimStruct` data structure for the S-Function block. Second, the code generator constructs a call through a function pointer to execute `mdlOutputs`, then `mdlOutputs`

calls `my_alg`. By inlining the call to your C/C++ algorithm, `my_alg`, you can eliminate both the `SimStruct` and the extra function call, thereby improving the efficiency and reducing the size of the generated code.

Inlining a wrapper S-function requires an *`sfunction`*`.tlc` file for the S-function (see the "Target Language Compiler" (Simulink Coder) for details). The TLC file must contain the function call to `my_alg`. The following figure shows the relationships between the algorithm, the wrapper S-function, and the *`sfunction`*`.tlc` file.



To inline this call, you have to place your function call in an *`sfunction`*`.tlc` file with the same name as the S-function (in this example, `wrapsfcn.tlc`). This causes the Target Language Compiler to override the default method of placing calls to your S-function in the generated code.

This is the `wrapsfcn.tlc` file that inlines `wrapsfcn.c`.

```
%% File    : wrapsfcn.tlc
%% Abstract:
%%      Example inlined tlc file for S-function wrapsfcn.c
%%

%implements "wrapsfcn" "C"

%% Function: BlockTypeSetup ===================================================
%% Abstract:
%%      Create function prototype in model.h as:
%%      "extern real_T my_alg(real_T u);"
%%
%function BlockTypeSetup(block, system) void
  %openfile buffer
    extern real_T my_alg(real_T u); /* This line is placed in wrapper.h */
  %closefile buffer
```

```
  %<LibCacheFunctionPrototype(buffer)>
%endfunction %% BlockTypeSetup

%% Function: Outputs ============================================================
%% Abstract:
%%      y = my_alg( u );
%%
%function Outputs(block, system) Output
  /* %<Type> Block: %<Name> */
  %assign u = LibBlockInputSignal(0, "", "", 0)
  %assign y = LibBlockOutputSignal(0, "", "", 0)
  %% PROVIDE THE CALLING STATEMENT FOR "algorithm"
  %% The following line is expanded and placed in mdlOutputs within wrapper.c
  %<y> = my_alg(%<u>);

%endfunction %% Outputs
```

The first section of this code inlines the `wrapsfcn` S-Function block and generates the code in C:

```
%implements "wrapsfcn" "C"
```

The next task is to tell the code generator that the routine `my_alg` needs to be declared external in the generated `wrapper.h` file for any `wrapsfcn` S-Function blocks in the model. You only need to do this once for all `wrapsfcn` S-Function blocks, so use the `BlockTypeSetup` function. In this function, you tell the Target Language Compiler to create a buffer and cache the `my_alg` as `extern` in the `wrapper.h` generated header file.

The final step is the inlining of the call to the function `my_alg`. This is done by the `Outputs` function. In this function, you access the block's input and output and place a direct call to `my_alg`. The call is embedded in `wrapper.c`.

The Inlined Code

The code generated when you inline your wrapper S-function is similar to the default generated code. The `mdlTerminate` function does not contain a call to an empty function and the `mdlOutputs` function now directly calls `my_alg`.

```
void mdlOutputs(int_T tid)
{
  /* Sin Block: <Root>/Sin */
  rtB.Sin = rtP.Sin.Amplitude *
    sin(rtP.Sin.Frequency * ssGetT(rtS) + rtP.Sin.Phase);

  /* S-Function Block: <Root>/S-Function */
  rtB.S_Function = my_alg(rtB.Sin); /* Inlined call to my_alg */

  /* Outport Block: <Root>/Out */
```

```
  rtY.Out = rtB.S_Function;
}
```

In addition, `wrapper.reg` does not create a child `SimStruct` for the S-function because the generated code is calling `my_alg` directly. This eliminates over 1 KB of memory usage.

Write Fully Inlined S-Functions

Using the example from "Write Wrapper S-Function and TLC Files" (Simulink Coder), you could eliminate the call to `my_alg` entirely by specifying the explicit code (that is, `2.0 * u`) in `wrapsfcn.tlc`. This is referred to as a *fully inlined S-function*. While this can improve performance, if you are working with a large amount of C/C++ code, this can be a lengthy task. In addition, you now have to maintain your algorithm in two places, the C/C++ S-function itself and the corresponding TLC file. However, the performance gains might outweigh the disadvantages. To inline the algorithm used in this example, in the `Outputs` section of your `wrapsfcn.tlc` file, instead of writing

```
%<y> = my_alg(%<u>);
```

use

```
%<y> = 2.0 * %<u>;
```

This is the code produced in `mdlOutputs`:

```
void mdlOutputs(int_T tid)
{
  /* Sin Block: <Root>/Sin */
  rtB.Sin = rtP.Sin.Amplitude *
    sin(rtP.Sin.Frequency * ssGetT(rtS) + rtP.Sin.Phase);

  /* S-Function Block: <Root>/S-Function */
  rtB.S_Function = 2.0 * rtB.Sin; /* Explicit embedding of algorithm */

  /* Outport Block: <Root>/Out */
  rtY.Out = rtB.S_Function;
}
```

The Target Language Compiler has replaced the call to `my_alg` with the algorithm itself.

Multiport S-Function

A more advanced multiport inlined S-function example is `sfun_multiport.c` and `sfun_multiport.tlc`. This S-function illustrates how to create a fully inlined TLC

file for an S-function that contains multiple ports. You might find that looking at this example helps you to understand fully inlined TLC files.

Write Fully Inlined S-Functions with mdlRTW Routine

You can inline more complex S-functions that use the S-function `mdlRTW` routine. The purpose of the `mdlRTW` routine is to provide the code generation process with more information about how the S-function is to be inlined, by creating a parameter record of a nontunable parameter for use with a TLC file. The `mdlRTW` routine does this by placing information in the *`model`*`.rtw` file. The `mdlRTW` function is described in the text file *`matlabroot`*`/simulink/src/sfuntmpl_doc.c`.

As an example of how to use the `mdlRTW` function, this topic discusses the steps you must take to create a direct-index lookup S-function. Lookup tables are collections of ordered data points of a function. Typically, these tables use some interpolation scheme to approximate values of the associated function between known data points. To incorporate the example lookup table algorithm into a Simulink model, the first step is to write an S-function that executes the algorithm in `mdlOutputs`. To produce the most efficient code, the next step is to create a corresponding TLC file to eliminate computational overhead and improve the speed of the lookup computations.

For your convenience, the Simulink product provides support for two general-purpose lookup 1-D and 2-D algorithms. You can use these algorithms as they are or create a custom lookup table S-function to fit your requirements. This topic illustrates how to create a 1-D lookup S-function, `sfun_directlook.c`, and its corresponding inlined `sfun_directlook.tlc` file (see “Target Language Compiler” (Simulink Coder) for more details). This 1-D direct-index lookup table example illustrates the following concepts that you need to know to create your own custom lookup tables:

- Error checking of S-function parameters
- Caching of information for the S-function that doesn't change during model execution
- How to use the `mdlRTW` function to customize the code generator to produce the optimal code for a given set of block parameters
- How to generate an inlined `TLC` file for an S-function in a combination of the fully inlined form and/or the wrapper form

- “S-Function RTWdata” on page 11-79
- “Direct-Index Lookup Table Algorithm” on page 11-79

S-Function RTWdata

There is a property of blocks called `RTWdata`, which can be used by the Target Language Compiler when inlining an S-function. `RTWdata` is a structure of character vectors that you can attach to a block. It is saved with the model and placed in the *`model`*`.rtw` file when generating code. For example, this set of MATLAB commands,

```
mydata.field1 = 'information for field1';
mydata.field2 = 'information for field2';
set_param(gcb,'RTWdata',mydata)
get_param(gcb,'RTWdata')
```

produces this result:

```
ans =

    field1: 'information for field1'
    field2: 'information for field2'
```

Inside the *`model`*`.rtw` file for the associated S-Function block is this information.

```
Block {
  Type                    "S-Function"
  RTWdata {
    field1                  "information for field1"
    field2                  "information for field2"
  }
```

Note: `RTWdata` is saved in the model file for S-functions that are not linked to a library. However, `RTWdata` is **not persistent** for S-Function blocks that are linked to a library.

Direct-Index Lookup Table Algorithm

The 1-D lookup table block provided in the Simulink library uses interpolation or extrapolation when computing outputs. This extra accuracy might not be required. In this example, you create a lookup table that directly indexes the output vector (*y*-data vector) based on the current input (*x*-data) point.

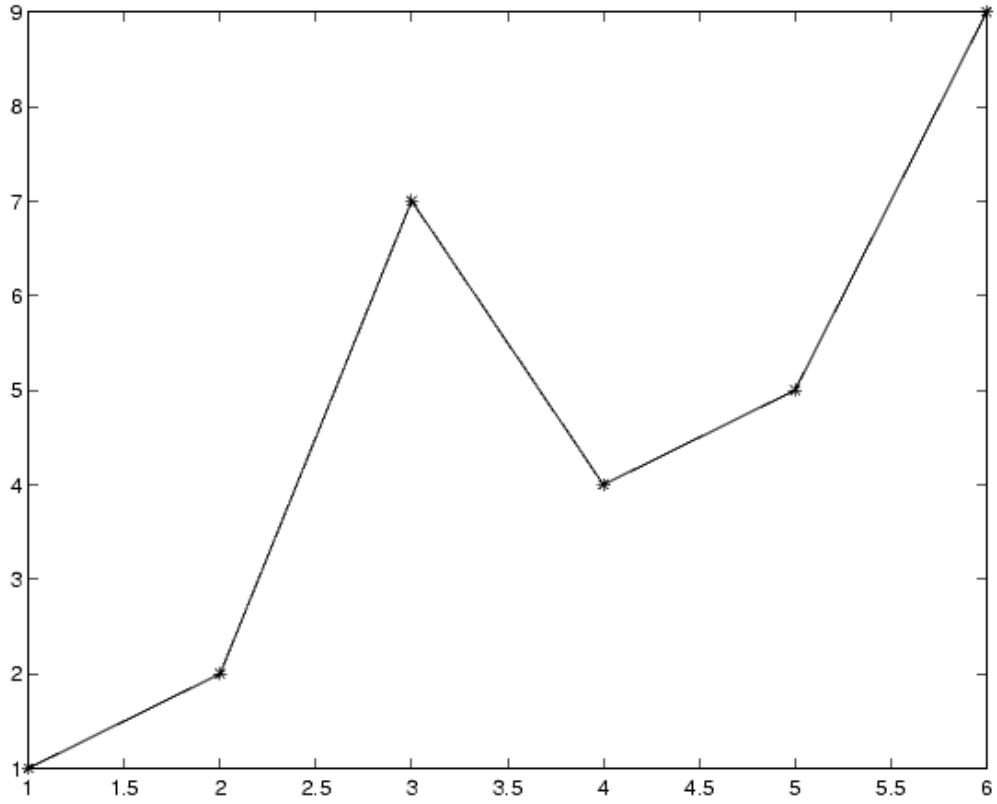
This direct 1-D lookup example computes an approximate solution *p(x)* to a partially known function *f(x)* at *x=x0*, given data point pairs *(x,y)* in the form of an *x*-data vector and a *y*-data vector. For a given data pair (for example, the *i'th* pair), *y_i = f(x_i)*. It is assumed that the *x*-data values are monotonically increasing. If *x0* is outside the range of the *x*-data vector, the first or last point is returned.

The parameters to the S-function are

```
XData, YData, XEvenlySpaced
```

`XData` and `YData` are double vectors of equal length representing the values of the unknown function. `XDataEvenlySpaced` is a scalar, `0.0` for false and `1.0` for true. If the `XData` vector is evenly spaced, `XDataEvenlySpaced` is `1.0` and more efficient code is generated.

The following graph shows how the parameters `XData=[1:6]` and `YData=[1,2,7,4,5,9]` are handled. For example, if the input (*x*-value) to the S-Function block is 3, the output (*y*-value) is 7.



Direct-Index Lookup Table Example

This topic shows how to improve the lookup table by inlining a direct-index S-function with a TLC file. This direct-index lookup table S-function does not require a TLC file. Here the example uses a TLC file for the direct-index lookup table S-function to reduce the code size and increase efficiency of the generated code.

Implementation of the direct-index algorithm with inlined TLC file requires the S-function main module, `sfun_directlook.c`, and a corresponding `lookup_index.c` module. The `lookup_index.c` module contains the `GetDirectLookupIndex` function that is used to locate the index in the `XData` for the current `x` input value when the `XData` is unevenly spaced. The `GetDirectLookupIndex` routine is called from both the S-function and the generated code. Here the example uses the wrapper concept for sharing C/C++ code between Simulink MEX-files and the generated code.

If the `XData` is evenly spaced, then both the S-function main module and the generated code contain the lookup algorithm (not a call to the algorithm) to compute the *y*-value of a given *x*-value, because the algorithm is short. This illustrates the use of a fully inlined S-function for generating optimal code.

The inlined TLC file, which either performs a wrapper call or embeds the optimal C/C++ code, is `sfun_directlook.tlc` (see the example in “mdlRTW Usage” on page 11-82).

Error Handling

In this example, the `mdlCheckParameters` routine verifies that

- The new parameter settings are valid.
- `XData` and `YData` are vectors of the same length containing real finite numbers.
- `XDataEvenlySpaced` is a scalar.
- The `XData` vector is a monotonically increasing vector and evenly spaced.

The `mdlInitializeSizes` function explicitly calls `mdlCheckParameters` after it verifies the number of parameters passed to the S-function. After the Simulink engine calls `mdlInitializeSizes`, it then calls `mdlCheckParameters` whenever you change the parameters or there is a need to reevaluate them.

User Data Caching

The `mdlStart` routine shows how to cache information that does not change during the simulation (or while the generated code is executing). The example caches the value

of the `XDataEvenlySpaced` parameter in `UserData`, a field of the `SimStruct`. The following line in `mdlInitializeSizes` tells the Simulink engine to disallow changes to `XDataEvenlySpaced`.

```
ssSetSFcnParamTunable(S, iParam, SS_PRM_NOT_TUNABLE);
```

During execution, `mdlOutputs` accesses the value of `XDataEvenlySpaced` from `UserData` rather than calling the `mxGetPr` MATLAB API function.

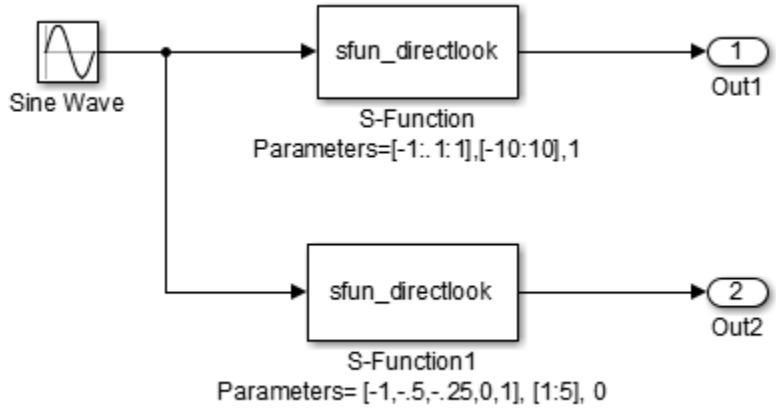
mdlRTW Usage

The code generator calls the `mdlRTW` routine while generating the *`model`*`.rtw` file. To produce optimal code for your Simulink model, you can add information to the *`model`*`.rtw` file about the mode in which your S-Function block is operating.

The following example adds parameter settings to the *`model`*`.rtw` file. The parameter settings do not change during execution. In this case, the `XDataEvenlySpaced` S-function parameter cannot change during execution (`ssSetSFcnParamTunable` was specified as false (`0`) for it in `mdlInitializeSizes`). The example writes it out as a parameter setting (`XSpacing`) using the function `ssWriteRTWParamSettings`.

Because `xData` and `yData` are registered as run-time parameters in `mdlSetWorkWidths`, the code generator handles writing to the *`model`*`.rtw` file automatically.

Before examining the S-function and the inlined TLC file, consider the generated code for the following model.



The model uses evenly spaced `XData` in the top S-Function block and unevenly spaced `XData` in the bottom S-Function block. When creating this model, you need to specify the following for each S-Function block.

```
set_param(`sfun_directlook_ex/S-Function','SFunctionModules','lookup_index')
set_param(`sfun_directlook_ex/S-Function1','SFunctionModules','lookup_index')
```

This informs the build process to use the module `lookup_index.c` when creating the executable.

When generating code for this model, the code generator uses the S-function `mdlRTW` method to generate a *`model`*`.rtw` file with the value `EvenlySpaced` for the `XSpacing` parameter for the top S-Function block, and the value `UnEvenlySpaced` for the `XSpacing` parameter for the bottom S-Function block. The TLC-file uses the value of `XSpacing` to determine what algorithm to include in the generated code. The generated code contains the lookup algorithm when the `XData` is evenly spaced, but calls the `GetDirectLookupIndex` routine when the `XData` is unevenly spaced. The generated *`model`*`.c` or *`model`*`.cpp` code for the lookup table example model is similar to the following:

```
/*
 * sfun_directlook_ex.c
 *
 * Code generation for Simulink model
 * "sfun_directlook_ex.slx".
 *
...
 */

#include "sfun_directlook_ex.h"
#include "sfun_directlook_ex_private.h"

/* External outputs (root outports fed by signals with auto storage) */
ExtY_sfun_directlook_ex_T sfun_directlook_ex_Y;

/* Real-time model */
RT_MODEL_sfun_directlook_ex_T sfun_directlook_ex_M_;
RT_MODEL_sfun_directlook_ex_T *const sfun_directlook_ex_M =
  &sfun_directlook_ex_M_;

/* Model output function */
void sfun_directlook_ex_output(void)
{
  /* local block i/o variables */
  real_T rtb_SFunction;
  real_T rtb_SFunction1;

  /* Sin: '<Root>/Sine Wave' */
  rtb_SFunction1 = sin(sfun_directlook_ex_M->Timing.t[0]);

/* Code that is inlined for the top S-function block in the
```

```
   * sfun_directlook_ex model
   */
  /* S-Function (sfun_directlook): '<Root>/S-Function' */
  {
    const real_T *xData = sfun_directlook_ex_ConstP.SFunction_XData;
    const real_T *yData = sfun_directlook_ex_ConstP.SFunction_YData;
    real_T spacing = xData[1] - xData[0];
    if (rtb_SFunction1 <= xData[0] ) {
      rtb_SFunction = yData[0];
    } else if (rtb_SFunction1 >= yData[20] ) {
      rtb_SFunction = yData[20];
    } else {
      int_T idx = (int_T)( ( rtb_SFunction1 - xData[0] ) / spacing );
      rtb_SFunction = yData[idx];
    }
  }

  /* Outport: '<Root>/Out1' */
  sfun_directlook_ex_Y.Out1 = rtb_SFunction;

/* Code that is inlined for the bottom S-function block in the
   * sfun_directlook_ex model
   */
  /* S-Function (sfun_directlook): '<Root>/S-Function1' */
  {
    const real_T *xData = sfun_directlook_ex_ConstP.SFunction1_XData;
    const real_T *yData = sfun_directlook_ex_ConstP.SFunction1_YData;
    int_T idx;
    idx = GetDirectLookupIndex(xData, 5, rtb_SFunction1);
    rtb_SFunction1 = yData[idx];
  }

  /* Outport: '<Root>/Out2' */
  sfun_directlook_ex_Y.Out2 = rtb_SFunction1;
}

/* Model update function */
void sfun_directlook_ex_update(void)
{
  /* signal main to stop simulation */
  {                                    /* Sample time: [0.0s, 0.0s] */
    if ((rtmGetTFinal(sfun_directlook_ex_M)!=-1) &&
        !((rtmGetTFinal(sfun_directlook_ex_M)-sfun_directlook_ex_M->Timing.t[0])
          > sfun_directlook_ex_M->Timing.t[0] * (DBL_EPSILON))) {
      rtmSetErrorStatus(sfun_directlook_ex_M, "Simulation finished");
    }
  }

  /* Update absolute time for base rate */
  /* The "clockTick0" counts the number of times the code of this task has
   * been executed. The absolute time is the multiplication of "clockTick0"
   * and "Timing.stepSize0". Size of "clockTick0" ensures timer will not
   * overflow during the application lifespan selected.
   * Timer of this task consists of two 32 bit unsigned integers.
   * The two integers represent the low bits Timing.clockTick0 and the high bits
```

```
   * Timing.clockTickH0. When the low bit overflows to 0, the high bits increment.
   */
  if (!(++sfun_directlook_ex_M->Timing.clockTick0)) {
    ++sfun_directlook_ex_M->Timing.clockTickH0;
  }

  sfun_directlook_ex_M->Timing.t[0] = sfun_directlook_ex_M->Timing.clockTick0 *
    sfun_directlook_ex_M->Timing.stepSize0 +
    sfun_directlook_ex_M->Timing.clockTickH0 *
    sfun_directlook_ex_M->Timing.stepSize0 * 4294967296.0;
}
...
```

matlabroot/toolbox/simulink/simdemos/simfeatures/src/sfun_directlook.c

```
/*
* File    :  sfun_directlook.c
 * Abstract:
 *
 *      Direct 1-D lookup. Here we are trying to compute an approximate
 *      solution, p(x) to an unknown function f(x) at x=x0, given data point
 *      pairs (x,y) in the form of a x data vector and a y data vector. For a
 *      given data pair (say the i'th pair), we have y_i = f(x_i). It is
 *      assumed that the x data values are monotonically increasing.  If the
 *      x0 is outside of the range of the x data vector, then the first or
 *      last point will be returned.
 *
 *      This function returns the "nearest" y0 point for a given x0.
 *      Interpolation is not performed.
 *
 *      The S-function parameters are:
 *        XData              - double vector
 *        YData              - double vector
 *        XDataEvenlySpacing - double scalar 0 (false) or 1 (true)
 *        The third parameter cannot be changed during simulation.
 *
 *      To build:
 *        mex sfun_directlook.c lookup_index.c
 *
 * Copyright 1990-2012 The MathWorks, Inc.
 */

#define S_FUNCTION_NAME  sfun_directlook
#define S_FUNCTION_LEVEL 2


#include <math.h>
#include "simstruc.h"
#include <float.h>

/* use utility function IsRealVect() */
#if defined(MATLAB_MEX_FILE)
#include "sfun_slutils.h"
#endif

/*================*
```

```
 * Build checking *
 *================*/
#if !defined(MATLAB_MEX_FILE)
/*
 * This file cannot be used directly with Simulink Coder. However,
 * this S-function does work with Simulink Coder via
 * the Target Language Compiler technology. See matlabroot/
 * toolbox/simulink/simdemos/simfeatures/tlc_c/sfun_directlook.tlc
 * for the C version
 */
# error This_file_can_be_used_only_during_simulation_inside_Simulink
#endif


/*=========*
 * Defines *
 *=========*/

#define XVECT_PIDX             0
#define YVECT_PIDX             1
#define XDATAEVENLYSPACED_PIDX 2
#define NUM_PARAMS             3

#define XVECT(S)             ssGetSFcnParam(S,XVECT_PIDX)
#define YVECT(S)             ssGetSFcnParam(S,YVECT_PIDX)
#define XDATAEVENLYSPACED(S) ssGetSFcnParam(S,XDATAEVENLYSPACED_PIDX)


/*==============*
 * misc defines *
 *==============*/
#if !defined(TRUE)
#define TRUE  1
#endif
#if !defined(FALSE)
#define FALSE 0
#endif

/*===========*
 * typedef's *
 *===========*/

typedef struct SFcnCache_tag {
    boolean_T evenlySpaced;
} SFcnCache;

/*====================================================================*
 * Prototype define for the function in separate file lookup_index.c *
 *====================================================================*/
extern int_T GetDirectLookupIndex(const real_T *x, int_T xlen, real_T u);


/*====================*
 * S-function methods *
 *====================*/
```

```
#define MDL_CHECK_PARAMETERS           /* Change to #undef to remove function */
#if defined(MDL_CHECK_PARAMETERS) && defined(MATLAB_MEX_FILE)
/* Function: mdlCheckParameters =============================================
 * Abstract:
 *    This routine will be called after mdlInitializeSizes, whenever
 *    parameters change or get re-evaluated. The purpose of this routine is
 *    to verify  the new parameter settings.
 *
 *    You should add a call to this routine from mdlInitalizeSizes
 *    to check the parameters. After setting your sizes elements, you should:
 *       if (ssGetSFcnParamsCount(S) == ssGetNumSFcnParams(S)) {
 *           mdlCheckParameters(S);
 *       }
 */
static void mdlCheckParameters(SimStruct *S)
{

    if (!IsRealVect(XVECT(S))) {
        ssSetErrorStatus(S,"1st, X-vector parameter must be a real finite "
                           " vector");
        return;
    }

    if (!IsRealVect(YVECT(S))) {
        ssSetErrorStatus(S,"2nd, Y-vector parameter must be a real finite "
                           "vector");
        return;
    }

    /*
     * Verify that the dimensions of X and Y are the same.
     */
    if (mxGetNumberOfElements(XVECT(S)) != mxGetNumberOfElements(YVECT(S)) ||
        mxGetNumberOfElements(XVECT(S)) == 1) {
        ssSetErrorStatus(S,"X and Y-vectors must be of the same dimension "
                           "and have at least two elements");
        return;
    }

    /*
     * Verify we have a valid XDataEvenlySpaced parameter.
     */
    if ((!mxIsNumeric(XDATAEVENLYSPACED(S)) &&
          !mxIsLogical(XDATAEVENLYSPACED(S))) ||
        mxIsComplex(XDATAEVENLYSPACED(S)) ||
        mxGetNumberOfElements(XDATAEVENLYSPACED(S)) != 1) {
        ssSetErrorStatus(S,"3rd, X-evenly-spaced parameter must be logical
scalar");
        return;
    }

    /*
     * Verify x-data is correctly spaced.
```

```
     */
    {
        size_t     i;
        boolean_T spacingEqual;
        real_T    *xData = mxGetPr(XVECT(S));
        size_t     numEl  = mxGetNumberOfElements(XVECT(S));

        /*
         * spacingEqual is TRUE if user  XDataEvenlySpaced
         */
        spacingEqual = (mxGetScalar(XDATAEVENLYSPACED(S)) != 0.0);

        if (spacingEqual) {    /* XData is 'evenly-spaced' */
            boolean_T badSpacing = FALSE;
            real_T    spacing    = xData[1] - xData[0];
            real_T    space;

            if (spacing <= 0.0) {
                badSpacing = TRUE;
            } else {
                real_T eps = DBL_EPSILON;

                for (i = 2; i < numEl; i++) {
                    space = xData[i] - xData[i-1];
                    if (space <= 0.0 ||
                        fabs(space-spacing) >= 128.0*eps*spacing ){
                        badSpacing = TRUE;
                        break;
                    }
                }
            }

            if (badSpacing) {
                ssSetErrorStatus(S,"X-vector must be an evenly spaced "
                                 "strictly monotonically increasing vector");
                return;
            }
        } else {     /* XData is 'unevenly-spaced' */
            for (i = 1; i < numEl; i++) {
                if (xData[i] <= xData[i-1]) {
                    ssSetErrorStatus(S,"X-vector must be a strictly "
                                       "monotonically increasing vector");
                    return;
                }
            }
        }
    }
}
#endif /* MDL_CHECK_PARAMETERS */


/* Function: mdlInitializeSizes ===============================================
 * Abstract:
 *    The sizes information is used by Simulink to determine the S-function
```

```
 *    block's characteristics (number of inputs, outputs, states, and so on).
 */
static void mdlInitializeSizes(SimStruct *S)
{
    ssSetNumSFcnParams(S, NUM_PARAMS);  /* Number of expected parameters */

    /*
     * Check parameters passed in, providing the correct number was specified
     * in the S-function dialog box. If an incorrect number of parameters
     * was specified, Simulink will detect the error since ssGetNumSFcnParams
     * and ssGetSFcnParamsCount will differ.
     *   ssGetNumSFcnParams   - This sets the number of parameters your
     *                          S-function expects.
     *   ssGetSFcnParamsCount - This is the number of parameters entered by
     *                          the user in the Simulink S-function dialog box.
     */
#if defined(MATLAB_MEX_FILE)
    if (ssGetNumSFcnParams(S) == ssGetSFcnParamsCount(S)) {
        mdlCheckParameters(S);
        if (ssGetErrorStatus(S) != NULL) {
            return;
        }
    } else {
        return; /* Parameter mismatch will be reported by Simulink */
    }
#endif

    {
        int iParam = 0;
        int nParam = ssGetNumSFcnParams(S);

        for ( iParam = 0; iParam < nParam; iParam++ )
        {
            switch ( iParam )
            {
              case XDATAEVENLYSPACED_PIDX:

                ssSetSFcnParamTunable( S, iParam, SS_PRM_NOT_TUNABLE );
                break;

              default:
                ssSetSFcnParamTunable( S, iParam, SS_PRM_TUNABLE );
                break;
            }
        }
    }

    ssSetNumContStates(S, 0);
    ssSetNumDiscStates(S, 0);

    if (!ssSetNumInputPorts(S, 1)) return;
    ssSetInputPortWidth(S, 0, DYNAMICALLY_SIZED);
    ssSetInputPortDirectFeedThrough(S, 0, 1);

    ssSetInputPortOptimOpts(S, 0, SS_REUSABLE_AND_LOCAL);
```

```
    ssSetInputPortOverWritable(S, 0, TRUE);

    if (!ssSetNumOutputPorts(S, 1)) return;
    ssSetOutputPortWidth(S, 0, DYNAMICALLY_SIZED);

    ssSetOutputPortOptimOpts(S, 0, SS_REUSABLE_AND_LOCAL);

    ssSetNumSampleTimes(S, 1);

    ssSetOptions(S,
                 SS_OPTION_WORKS_WITH_CODE_REUSE |
                 SS_OPTION_EXCEPTION_FREE_CODE |
                 SS_OPTION_USE_TLC_WITH_ACCELERATOR);

} /* mdlInitializeSizes */


/* Function: mdlInitializeSampleTimes =========================================
 * Abstract:
 *    The lookup inherits its sample time from the driving block.
 */
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
    ssSetModelReferenceSampleTimeDefaultInheritance(S);
} /* end mdlInitializeSampleTimes */

/* Function: mdlSetWorkWidths =================================================
 * Abstract:
 *    Set up the [X,Y] data as run-time parameters
 *    that is, these values can be changed during execution.
 */
#define MDL_SET_WORK_WIDTHS
static void mdlSetWorkWidths(SimStruct *S)
{
    const char_T    *rtParamNames[] = {"XData","YData"};
    ssRegAllTunableParamsAsRunTimeParams(S, rtParamNames);
}


#define MDL_START                      /* Change to #undef to remove function */
#if defined(MDL_START)
/* Function: mdlStart =======================================================
 * Abstract:
 *      Here we cache the state (true/false) of the XDATAEVENLYSPACED parameter.
 *      We do this primarily to illustrate how to "cache" parameter values (or
 *      information which is computed from parameter values) which do not change
 *      for the duration of the simulation (or in the generated code). In this
 *      case, rather than repeated calls to mxGetPr, we save the state once.
 *      This results in a slight increase in performance.
 */
static void mdlStart(SimStruct *S)
{
    SFcnCache *cache = malloc(sizeof(SFcnCache));
```

```
    if (cache == NULL) {
        ssSetErrorStatus(S,"memory allocation error");
        return;
    }

    ssSetUserData(S, cache);

    if (mxGetScalar(XDATAEVENLYSPACED(S)) != 0.0){
        cache->evenlySpaced = TRUE;
    }else{
        cache->evenlySpaced = FALSE;
    }

}
#endif /*  MDL_START */



/* Function: mdlOutputs =======================================================
 * Abstract:
 *    In this function, you compute the outputs of your S-function
 *    block. Generally outputs are placed in the output vector, ssGetY(S).
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
    SFcnCache         *cache = ssGetUserData(S);
    real_T            *xData = mxGetPr(XVECT(S));
    real_T            *yData = mxGetPr(YVECT(S));
    InputRealPtrsType uPtrs  = ssGetInputPortRealSignalPtrs(S,0);
    real_T            *y     = ssGetOutputPortRealSignal(S,0);
    size_t             ny     = ssGetOutputPortWidth(S,0);
    size_t             xLen   = mxGetNumberOfElements(XVECT(S));
    size_t             i;

    /*
     * When the XData is evenly spaced, we use the direct lookup algorithm
     * to calculate the lookup
     */
    if (cache->evenlySpaced) {
        real_T spacing = xData[1] - xData[0];
        for (i = 0; i < ny; i++) {
            real_T u = *uPtrs[i];

            if (u <= xData[0]) {
                y[i] = yData[0];
            } else if (u >= xData[xLen-1]) {
                y[i] = yData[xLen-1];
            } else {
                int_T idx = (int_T)((u - xData[0])/spacing);
                y[i] = yData[idx];
            }
        }
    } else {
        /*
```

```
         * When the XData is unevenly spaced, we use a bisection search to
         * locate the lookup index.
         */
        for (i = 0; i < ny; i++) {
            int_T idx = GetDirectLookupIndex(xData,xLen,*uPtrs[i]);
            y[i] = yData[idx];
        }
    }

} /* end mdlOutputs */



/* Function: mdlTerminate =====================================================
 * Abstract:
 *    Free the cache which was allocated in mdlStart.
 */
static void mdlTerminate(SimStruct *S)
{
    SFcnCache *cache = ssGetUserData(S);
    if (cache != NULL) {
        free(cache);
    }
} /* end mdlTerminate */



#define MDL_RTW                      /* Change to #undef to remove function */
#if defined(MDL_RTW) && (defined(MATLAB_MEX_FILE) || defined(NRT))
/* Function: mdlRTW ===========================================================
 * Abstract:
 *    This function is called when Simulink Coder is generating the
 *    model.rtw file. In this routine, you can call the following functions
 *    which add fields to the model.rtw file.
 *
 *    Important! Since this S-function has this mdlRTW method, it is required
 *    to have a corresponding .tlc file so as to work with Simulink Coder. See the
 *    sfun_directlook.tlc in matlabroot/toolbox/simulink/simdemos/simfeatures/tlc_c/.
 */
static void mdlRTW(SimStruct *S)
{
    /*
     * Write out the spacing setting as a param setting, that is, this cannot be
     * changed during execution.
     */
    {
        boolean_T even = (mxGetScalar(XDATAEVENLYSPACED(S)) != 0.0);

        if (!ssWriteRTWParamSettings(S, 1,
                                     SSWRITE_VALUE_QSTR,
                                     "XSpacing",
                                     even ? "EvenlySpaced" : "UnEvenlySpaced")){
            return;/* An error occurred which will be reported by Simulink */
        }
    }
```

```
}
#endif /* MDL_RTW */


/*============================*
 * Required S-function trailer *
 *============================*/

#ifdef  MATLAB_MEX_FILE    /* Is this file being compiled as a MEX-file? */
#include "simulink.c"      /* MEX-file interface mechanism */
#else
#include "cg_sfun.h"       /* Code generation registration function */
#endif


/* [EOF] sfun_directlook.c */
```

matlabroot/toolbox/simulink/simdemos/simfeatures/src/lookup_index.c

```
/* File    : lookup_index.c
* Abstract:
 *
 *     Contains a routine used by the S-function sfun_directlookup.c to
 *     compute the index in a vector for a given data value.
 *
 *  Copyright 1990-2014 The MathWorks, Inc.
 */
#ifdef MATLAB_MEX_FILE
#include <tmwtypes.h>
#else
#include "rtwtypes.h"
#endif

/*
 * Function: GetDirectLookupIndex ==============================================
 * Abstract:
 *     Using a bisection search to locate the lookup index when the x-vector
 *     isn't evenly spaced.
 *
 *     Inputs:
 *        *x   : Pointer to table, x[0] ....x[xlen-1]
 *        xlen : Number of values in xtable
 *        u    : input value to look up
 *
 *     Output:
 *        idx  : the index into the table such that:
 *               if u is negative
 *                  x[idx] <= u < x[idx+1]
 *               else
 *                  x[idx] < u <= x[idx+1]
 */
int_T GetDirectLookupIndex(const real_T *x, int_T xlen, real_T u)
{
    int_T idx    = 0;
    int_T bottom = 0;
    int_T top    = xlen-1;
```

```
    /*
     * Deal with the extreme cases first:
     *
     *  i] u <= x[bottom] then idx = bottom
     * ii] u >= x[top] then idx = top-1
     *
     */
    if (u <= x[bottom]) {
        return(bottom);
    } else if (u >= x[top]) {
        return(top);
    }

    /*
     * We have: x[bottom] < u < x[top], onward
     * with search for the  index ...
     */
    for (;;) {
        idx = (bottom + top)/2;
        if (u < x[idx]) {
            top = idx;
        } else if (u > x[idx+1]) {
            bottom = idx + 1;
        } else {
            /*
             * We have:  x[idx] <= u <= x[idx+1], only need
             * to do two more checks and we have the answer
             */
            if (u < 0) {
                /*
                 * We want right continuity, that is,
                 * if u == x[idx+1]
                 *    then x[idx+1] <= u < x[idx+2]
                 * else    x[idx  ] <= u < x[idx+1]
                 */
                return( (u == x[idx+1]) ? (idx+1) : idx);
            } else {
                /*
                 * We want left continuity, that is,
                 * if u == x[idx]
                 *    then x[idx-1] < u <= x[idx  ]
                 * else    x[idx  ] < u <= x[idx+1]
                 */
                return( (u == x[idx]) ? (idx-1) : idx);
            }
        }
    }
} /* end GetDirectLookupIndex */

/* [EOF] lookup_index.c */
```

matlabroot/toolbox/simulink/simdemos/simfeatures/tlc_c/sfun_directlook.tlc

```
%% File    : sfun_directlook.tlc
%% Abstract:
```

```
%%      Level-2 S-function sfun_directlook block target file.
%%      It is using direct lookup algorithm without interpolation
%%
%% Copyright 1990-2010 The MathWorks, Inc.
%%


%implements "sfun_directlook" "C"

%% Function: BlockTypeSetup ===================================================
%% Abstract:
%%     Place include and function prototype in the model's header file.
%%
%function BlockTypeSetup(block, system) void

  %% To add this external function's prototype in the header of the generated
  %% file.
  %%
  %openfile buffer
  extern int_T GetDirectLookupIndex(const real_T *x, int_T xlen, real_T u);
  %closefile buffer

  %<LibCacheFunctionPrototype(buffer)>

%endfunction

%% Function: mdlOutputs =======================================================
%% Abstract:
%%      Direct 1-D lookup table S-function example.
%%      Here we are trying to compute an approximate solution, p(x) to an
%%      unknown function f(x) at x=x0, given data point pairs (x,y) in the
%%      form of a x data vector and a y data vector. For a given data pair
%%      (say the i'th pair), we have y_i = f(x_i). It is assumed that the x
%%      data values are monotonically increasing.  If the first or last x is
%%      outside of the range of the x data vector, then the first or last
%%      point will be returned.
%%
%%      This function returns the "nearest" y0 point for a given x0.
%%      Interpolation is not performed.
%%
%%      The S-function parameters are:
%%        XData
%%        YData
%%        XEvenlySpaced: 0 or 1
%%      The third parameter cannot be changed during execution and is
%%      written to the model.rtw file in XSpacing filed of the SFcnParamSettings
%%      record as "EvenlySpaced" or "UnEvenlySpaced". The first two parameters
%%      can change during execution and show up in the parameter vector.
%%
%function Outputs(block, system) Output
  /* %<Type> Block: %<Name> */
  {
    %assign rollVars = ["U", "Y"]
    %%
    %% Load XData and YData as local variables
```

```
    %%
    const real_T *xData  = %<LibBlockParameterAddr(XData, "", "", 0)>;
    const real_T *yData  = %<LibBlockParameterAddr(YData, "", "", 0)>;
    %assign xDataLen = SIZE(XData.Value, 1)
    %%
    %% When the XData is evenly spaced, we use the direct lookup algorithm
    %% to locate the lookup index.
    %%
    %if SFcnParamSettings.XSpacing == "EvenlySpaced"
      real_T spacing = xData[1] - xData[0];

      %roll idx = RollRegions, lcv = RollThreshold, block, "Roller", rollVars
        %assign u = LibBlockInputSignal(0, "", lcv, idx)
        %assign y = LibBlockOutputSignal(0, "", lcv, idx)
        if ( %<u> <= xData[0] ) {
          %<y> = yData[0];
        } else if ( %<u> >= yData[%<xDataLen-1>] ) {
          %<y> = yData[%<xDataLen-1>];
        } else {
          int_T idx = (int_T)( ( %<u> - xData[0] ) / spacing );
          %<y> = yData[idx];
        }
        %%
        %% Generate an empty line if we are not rolling,
        %% so that it looks nice in the generated code.
        %%
        %if lcv == ""

        %endif
      %endroll
    %else
      %% When the XData is unevenly spaced, we use a bisection search to
      %% locate the lookup index.
      int_T idx;

      %assign xDataAddr = LibBlockParameterAddr(XData, "", "", 0)
      %roll idx = RollRegions, lcv = RollThreshold, block, "Roller", rollVars
        %assign u = LibBlockInputSignal(0, "", lcv, idx)
        idx =  GetDirectLookupIndex(xData, %<xDataLen>, %<u>);
        %assign y = LibBlockOutputSignal(0, "", lcv, idx)
        %<y> = yData[idx];
        %%
        %% Generate an empty line if we are not rolling,
        %% so that it looks nice in the generated code.
        %%
        %if lcv == ""

        %endif
      %endroll
    %endif
  }
%endfunction
%% EOF: sfun_directlook.tlc
```

Guidelines for Writing Inlined S-Functions

- Consider using the block property `RTWdata` (see "S-Function RTWdata" (Simulink Coder)). This property is a structure of character vectors that you can associate with a block. The code generator saves the structure with the model in the *`model`*`.rtw` file and makes the `.rtw` file more readable. For example, suppose you enter the following commands in the MATLAB Command Window:

  ```
  mydata.field1 = 'information for field1';
  mydata.field2 = 'information for field2';
  set_param(sfun_block, 'RTWdata', mydata);
  ```

  The `.rtw` file that the code generator produces for the block includes the comments specified in the structure `mydata`.
- Consider using the `mdlRTW` function to inline your C MEX S-function in the generated code. This is useful when you want to
  - Rename tunable parameters in the generated code
  - Introduce nontunable parameters into a TLC file

S-Functions That Support Expression Folding

About S-Functions that Support Expression Folding

This topic describes how you can take advantage of expression folding to increase the efficiency of code generated by your own inlined S-Function blocks, by calling macros provided in the S-Function API. This topic assumes that you are familiar with:

- Writing inlined S-functions (see "S-Function Basics" (Simulink)).
- "Target Language Compiler" (Simulink Coder)

The S-Function API lets you specify whether a given S-Function block should nominally accept expressions at a given input port. A block should not always accept expressions. For example, if the address of the signal at the input is used, expressions should not be accepted at that input, because it is not possible to take the address of an expression.

The S-Function API also lets you specify whether an expression can represent the computations associated with a given output port. When you request an expression at a block's input or output port, the Simulink engine determines whether or not it can honor that request, given the block's context. For example, the engine might deny a block's request to output an expression if the destination block does not accept expressions at its input, if the destination block has an update function, or if multiple output destinations exist.

The decision to honor or deny a request to output an expression can also depend on the category of output expression the block uses (see "Categories of Output Expressions" on page 11-98).

The topics that follow explain

- When and how you can request that a block accept expressions at an input port
- When and how you can request that a block generate expressions at an outport
- The conditions under which the Simulink engine will honor or deny such requests

To take advantage of expression folding in your S-functions, you should understand when to request acceptance and generation of expressions for specific blocks. You do not have to understand the algorithm by which the Simulink engine chooses to accept or deny these requests. However, if you want to trace between the model and the generated code, it is helpful to understand some of the more common situations that lead to denial of a request.

Categories of Output Expressions

When you implement a C MEX S-function, you can specify whether the code corresponding to a block's output is to be generated as an expression. If the block generates an expression, you must specify that the expression is *constant*, *trivial*, or *generic*.

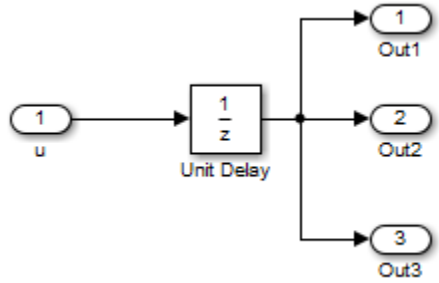
A *constant* output expression is a direct access to one of the block's parameters. For example, the output of a Constant block is defined as a constant expression because the output expression is simply a direct access to the block's `Value` parameter.

A *trivial* output expression is an expression that can be repeated, without a performance penalty, when the output port has multiple output destinations. For example, the output of a Unit Delay block is defined as a trivial expression because the output expression is simply a direct access to the block's state. Because the output expression does not have computations, it can be repeated more than once without degrading the performance of the generated code.

A *generic* output expression is an expression that should be assumed to have a performance penalty if repeated. As such, a generic output expression is not suitable for repeating when the output port has multiple output destinations. For instance, the output of a Sum block is a generic rather than a trivial expression because it is costly to recompute a Sum block output expression as an input to multiple blocks.

Examples of Trivial and Generic Output Expressions

Consider the following block diagram. The Delay block has multiple destinations, yet its output is designated as a trivial output expression, so that it can be used more than once without degrading the efficiency of the code.

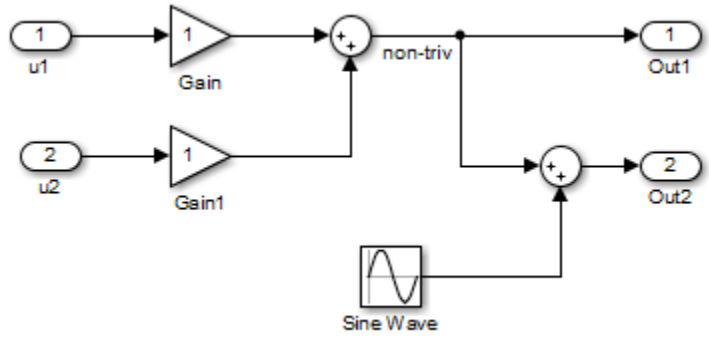


The following code excerpt shows code generated from the Unit Delay block in this block diagram. The three root outputs are directly assigned from the state of the Unit Delay block, which is stored in a field of the global data structure `rtDWork`. Since the assignment is direct, without expressions, there is no performance penalty associated with using the trivial expression for multiple destinations.

```
void MdlOutputs(int_T tid)
{
   ...
  /* Outport: <Root>/Out1 incorporates:

   *   UnitDelay: <Root>/Unit Delay */
  rtY.Out1 = rtDWork.Unit_Delay_DSTATE;

  /* Outport: <Root>/Out2 incorporates:
   *   UnitDelay: <Root>/Unit Delay */
  rtY.Out2 = rtDWork.Unit_Delay_DSTATE;

  /* Outport: <Root>/Out3 incorporates:
   *   UnitDelay: <Root>/Unit Delay */
  rtY.Out3 = rtDWork.Unit_Delay_DSTATE;
```

```
    ...
}
```

On the other hand, consider the Sum blocks in the following model:



The upper Sum block in the preceding model generates the signal labeled `non_triv`. Computation of this output signal involves two multiplications and an addition. If the Sum block's output were permitted to generate an expression even when the block had multiple destinations, the block's operations would be duplicated in the generated code. In the case illustrated, the generated expressions would proliferate to four multiplications and two additions. This would degrade the efficiency of the program. Accordingly the output of the Sum block is not allowed to be an expression because it has multiple destinations

The code generated for the previous block diagram shows how code is generated for Sum blocks with single and multiple destinations.

The Simulink engine does not permit the output of the upper Sum block to be an expression because the signal `non_triv` is routed to two output destinations. Instead, the result of the multiplication and addition operations is stored in a temporary variable `(rtb_non_triv)` that is referenced twice in the statements that follow, as seen in the code excerpt below.

In contrast, the lower Sum block, which has only a single output destination (`Out2`), does generate an expression.

```
void MdlOutputs(int_T tid)
{
  /* local block i/o variables */
  real_T rtb_non_triv;
```

```
  real_T rtb_Sine_Wave;

  /* Sum: <Root>/Sum incorporates:
   *   Gain: <Root>/Gain
   *   Inport: <Root>/u1
   *   Gain: <Root>/Gain1
   *   Inport: <Root>/u2
   *
   * Regarding <Root>/Gain:
   *   Gain value: rtP.Gain_Gain
   *
   * Regarding <Root>/Gain1:
   *   Gain value: rtP.Gain1_Gain
   */
  rtb_non_triv = (rtP.Gain_Gain * rtU.u1) + (rtP.Gain1_Gain *
rtU.u2);

  /* Outport: <Root>/Out1 */
  rtY.Out1 = rtb_non_triv;

  /* Sin Block: <Root>/Sine Wave */

  rtb_Sine_Wave = rtP.Sine_Wave_Amp *
 sin(rtP.Sine_Wave_Freq * rtmGetT(rtM_model) +
 rtP.Sine_Wave_Phase) + rtP.Sine_Wave_Bias;

  /* Outport: <Root>/Out2 incorporates:
   *   Sum: <Root>/Sum1
   */
  rtY.Out2 = (rtb_non_triv + rtb_Sine_Wave);
}
```

Specify the Category of an Output Expression

The S-Function API provides macros that let you declare whether an output of a block should be an expression, and if so, to specify the category of the expression. The following table specifies when to declare a block output to be a constant, trivial, or generic output expression.

Types of Output Expressions

Category of Expression	When to Use
Constant	Use only if block output is a direct memory access to a block parameter.
Trivial	Use only if block output is an expression that can appear multiple times in the code without reducing efficiency (for example, a direct memory access to a field of the `DWork` vector, or a literal).
Generic	Use if output is an expression, but not constant or trivial.

You must declare outputs as expressions in the `mdlSetWorkWidths` function using macros defined in the S-Function API. The macros have the following arguments:

- `SimStruct *S`: pointer to the block's `SimStruct`.
- `int idx:` zero-based index of the output port.
- `bool value:` pass in TRUE if the port generates output expressions.

The following macros are available for setting an output to be a constant, trivial, or generic expression:

- `void ssSetOutputPortConstOutputExprInRTW(SimStruct *S, int idx, bool value)`
- `void ssSetOutputPortTrivialOutputExprInRTW(SimStruct *S, int idx, bool value)`
- `void ssSetOutputPortOutputExprInRTW(SimStruct *S, int idx, bool value)`

The following macros are available for querying the status set by prior calls to the macros above:

- `bool ssGetOutputPortConstOutputExprInRTW(SimStruct *S, int idx)`
- `bool ssGetOutputPortTrivialOutputExprInRTW(SimStruct *S, int idx)`
- `bool ssGetOutputPortOutputExprInRTW(SimStruct *S, int idx)`

The set of generic expressions is a superset of the set of trivial expressions, and the set of trivial expressions is a superset of the set of constant expressions.

Therefore, when you query an output that has been set to be a constant expression with `ssGetOutputPortTrivialOutputExprInRTW`, it returns `True`. A constant expression is considered a trivial expression, because it is a direct memory access that can be repeated without degrading the efficiency of the generated code.

Similarly, an output that has been configured to be a constant or trivial expression returns `True` when queried for its status as a generic expression.

Acceptance or Denial of Requests for Input Expressions

A block can request that its output be represented in code as an expression. Such a request can be denied if the destination block cannot accept expressions at its input port. Furthermore, conditions independent of the requesting block and its destination blocks can prevent acceptance of expressions.

This topic discusses block-specific conditions under which requests for input expressions are denied. For information on other conditions that prevent acceptance of expressions, see “Denial of Block Requests to Output Expressions” on page 11-104.

A block should not be configured to accept expressions at its input port under the following conditions:

- The block must take the address of its input data. It is not possible to take the address of most types of input expressions.
- The code generated for the block references the input more than once (for example, the Abs or Max blocks). This would lead to duplication of a potentially complex expression and a subsequent degradation of code efficiency.

If a block refuses to accept expressions at an input port, then a block that is connected to that input port is not permitted to output a generic or trivial expression.

A request to output a constant expression is not denied, because there is no performance penalty for a constant expression, and the software can take the parameter’s address.

S-Function API to Specify Input Expression Acceptance

The S-Function API provides macros that let you:

- Specify whether a block input should accept nonconstant expressions (that is, trivial or generic expressions)
- Query whether a block input accepts nonconstant expressions

By default, block inputs do not accept nonconstant expressions.

You should call the macros in your `mdlSetWorkWidths` function. The macros have the following arguments:

- `SimStruct *S`: pointer to the block's SimStruct.
- `int idx:` zero-based index of the input port.
- `bool value:` pass in TRUE if the port accepts input expressions; otherwise pass in FALSE.

The macro available for specifying whether or not a block input should accept a nonconstant expression is as follows:

```
void ssSetInputPortAcceptExprInRTW(SimStruct *S, int portIdx, bool value)
```

The corresponding macro available for querying the status set by any prior calls to `ssSetInputPortAcceptExprInRTW` is as follows:

```
bool ssGetInputPortAcceptExprInRTW(SimStruct *S, int portIdx)
```

Denial of Block Requests to Output Expressions

Even after a specific block requests that it be allowed to generate an output expression, that request can be denied for generic reasons. These reasons include, but are not limited to

- The output expression is nontrivial, and the output has multiple destinations.
- The output expression is nonconstant, and the output is connected to at least one destination that does not accept expressions at its input port.
- The output is a test point.
- The output has been assigned an external storage class.
- The output must be stored using global data (for example is an input to a merge block or a block with states).
- The output signal is complex.

You do not need to consider these generic factors when deciding whether or not to utilize expression folding for a particular block. However, these rules can be helpful when you are examining generated code and analyzing cases where the expression folding optimization is suppressed.

Expression Folding in a TLC Block Implementation

To take advantage of expression folding, you must modify the TLC block implementation of an inlined S-Function such that it informs the Simulink engine whether it generates or accepts expressions at its

- Input ports, as explained in "S-Function API to Specify Input Expression Acceptance" on page 11-103.
- Output ports, as explained in "Categories of Output Expressions" on page 11-98.

This topic discusses required modifications to the TLC implementation.

Expression Folding Compliance

In the `BlockInstanceSetup` function of your S-function, register your block to be compliant with expression folding. Otherwise, expression folding requested or allowed at the block's outputs or inputs will be disabled, and temporary variables will be used.

To register expression folding compliance, call the TLC library function `LibBlockSetIsExpressionCompliant(block)`, which is defined in matlabroot/rtw/c/tlc/lib/utillib.tlc. For example:

```
%% Function: BlockInstanceSetup ===========================================
%%
%function BlockInstanceSetup(block, system) void
  %%
  %<LibBlockSetIsExpressionCompliant(block)>
  %%
%endfunction
```

You can conditionally disable expression folding at the inputs and outputs of a block by making the call to this function conditionally.

If you override one of the TLC block implementations provided by the code generator with your own implementation, you should not make the preceding call until you have updated your implementation, as described by the guidelines for expression folding in the following topics.

Output Expressions

The `BlockOutputSignal` function is used to generate code for a scalar output expression or one element of a nonscalar output expression. If your block outputs an expression, you should add a `BlockOutputSignal` function. The prototype of the `BlockOutputSignal` is

```
%function BlockOutputSignal(block,system,portIdx,ucv,lcv,idx,retType) void
```

The arguments to `BlockOutputSignal` are as follows:

- `block`: the record for the block for which an output expression is being generated
- `system`: the record for the system containing the block
- `portIdx`: zero-based index of the output port for which an expression is being generated
- `ucv`: user control variable defining the output element for which code is being generated
- `lcv`: loop control variable defining the output element for which code is being generated
- `idx`: signal index defining the output element for which code is being generated
- `retType`: character vector defining the type of signal access desired:

  `"Signal"` specifies the contents or address of the output signal.

`"SignalAddr"` specifies the address of the output signal

The `BlockOutputSignal` function returns a character vector for the output signal or address. The character vector should enforce the precedence of the expression by using opening and terminating parentheses, unless the expression consists of a function call. The address of an expression can only be returned for a constant expression; it is the address of the parameter whose memory is being accessed. The code implementing the `BlockOutputSignal` function for the Constant block is shown below.

```
%% Function: BlockOutputSignal ================================================
%% Abstract:
%%      Return the reference to the parameter.  This function *may*
%%      be used by Simulink when optimizing the Block IO data structure.
%%
%function BlockOutputSignal(block,system,portIdx,ucv,lcv,idx,retType) void
  %switch retType
    %case "Signal"
      %return LibBlockParameter(Value,ucv,lcv,idx)
    %case "SignalAddr"
      %return LibBlockParameterAddr(Value,ucv,lcv,idx)
    %default
      %assign errTxt = "Unsupported return type: %<retType>"
      %<LibBlockReportError(block,errTxt)>
  %endswitch
%endfunction
```

The code implementing the `BlockOutputSignal` function for the Relational Operator block is shown below.

```
%% Function: BlockOutputSignal ================================================
%% Abstract:
%%      Return an output expression.  This function *may*
%%      be used by Simulink when optimizing the Block IO data structure.
%%
%function BlockOutputSignal(block,system,portIdx,ucv,lcv,idx,retType) void
%switch retType
%case "Signal"
%assign logicOperator = ParamSettings.Operator
 %if ISEQUAL(logicOperator, "~=")
 %assign op = "!="
elseif ISEQUAL(logicOperator, "==") %assign op = "=="
  %else
%assign op = logicOperator
%endif
 %assign u0 = LibBlockInputSignal(0, ucv, lcv, idx)
%assign u1 = LibBlockInputSignal(1, ucv, lcv, idx)
  %return "(%<u0> %<op> %<u1>)"
 %default
 %assign errTxt = "Unsupported return type: %<retType>"
 %<LibBlockReportError(block,errTxt)>
%endswitch
```

```
%endfunction
```

Expression Folding for Blocks with Multiple Outputs

When a block has a single output, the `Outputs` function in the block's TLC file is called only if the output port is not an expression. Otherwise, the `BlockOutputSignal` function is called.

If a block has multiple outputs, the `Outputs` function is called if any output port is not an expression. The `Outputs` function should guard against generating code for output ports that are expressions. This is achieved by guarding sections of code corresponding to individual output ports with calls to `LibBlockOutputSignalIsExpr()`.

For example, consider an S-Function with two inputs and two outputs, where

- The first output, y0, is equal to two times the first input.
- The second output, y1, is equal to four times the second input.

The `Outputs` and `BlockOutputSignal` functions for the S-function are shown in the following code excerpt.

```
%% Function: BlockOutputSignal ================================================
%% Abstract:
%%      Return an output expression.  This function *may*
%%      be used by Simulink when optimizing the Block IO data structure.
%%
%function BlockOutputSignal(block,system,portIdx,ucv,lcv,idx,retType) void
%switch retType
%case "Signal"
   %assign u = LibBlockInputSignal(portIdx, ucv, lcv, idx)
 %case "Signal"
 %if portIdx == 0
  %return "(2 * %<u>)"
%elseif portIdx == 1
 %return "(4 * %<u>)"
%endif
%default
%assign errTxt = "Unsupported return type: %<retType>"
 %<LibBlockReportError(block,errTxt)>
%endswitch
%endfunction
%%
%% Function: Outputs ==========================================================
%% Abstract:
%%      Compute output signals of block
%%
%function Outputs(block,system) Output
%assign rollVars = ["U", "Y"]
%roll sigIdx = RollRegions, lcv = RollThreshold, block, "Roller", rollVars
%assign u0 = LibBlockInputSignal(0, "", lcv, sigIdx)
 %assign u1 = LibBlockInputSignal(1, "", lcv, sigIdx)
```

```
 %assign y0 = LibBlockOutputSignal(0, "", lcv, sigIdx)
 %assign y1 = LibBlockOutputSignal(1, "", lcv, sigIdx)
%if !LibBlockOutputSignalIsExpr(0)
%<y0> = 2 * %<u0>;
%endif
%if !LibBlockOutputSignalIsExpr(1)
 %<y1> = 4 * %<u1>;
%endif
%endroll
%endfunction
```

Comments for Blocks That Are Expression-Folding-Compliant

In the past, blocks preceded their outputs code with comments of the form

```
/* %<Type> Block: %<Name> */
```

When a block is expression-folding-compliant, the initial line shown above is generated automatically. You should not include the comment as part of the block's TLC implementation. Additional information should be registered using the `LibCacheBlockComment` function.

The `LibCacheBlockComment` function takes a character vector as an input, defining the body of the comment, except for the opening header, the final newline of a single or multiline comment, and the closing trailer.

The following TLC code illustrates registering a block comment. Note the use of the function `LibBlockParameterForComment`, which returns a character vector, suitable for a block comment, specifying the value of the block parameter.

```
%openfile commentBuf
  $c(*) Gain value: %<LibBlockParameterForComment(Gain)>
  %closefile commentBuf
  %<LibCacheBlockComment(block, commentBuf)>
```

S-Functions That Specify Port Scope and Reusability

You can use the following `SimStruct` macros in the `mdlInitializeSizes` method to specify the scope and reusability of the memory used for your S-function's input and output ports:

- `ssSetInputPortOptimOpts`: Specify the scope and reusability of the memory allocated to an S-function input port
- `ssSetOutputPortOptimOpts`: Specify the scope and reusability of the memory allocated to an S-function output port
- `ssSetInputPortOverWritable`: Specify whether one of your S-function's input ports can be overwritten by one of its output ports

- `ssSetOutputPortOverwritesInputPort`: Specify whether an output port can share its memory buffer with an input port

You declare an input or output as local or global, and indicate its reusability, by passing one of the following four options to the `ssSetInputPortOptimOpts` and `ssSetOutputPortOptimOpts` macros:

- `SS_NOT_REUSABLE_AND_GLOBAL`: Indicates that the input and output ports are stored in separate memory locations in the global block input and output structure
- `SS_NOT_REUSABLE_AND_LOCAL`: Indicates that the code generator can declare individual local variables for the input and output ports
- `SS_REUSABLE_AND_LOCAL`: Indicates that the code generator can reuse a single local variable for these input and output ports
- `SS_REUSABLE_AND_GLOBAL`: Indicates that these input and output ports are stored in a single element in the global block input and output structure

---

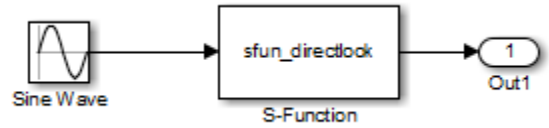
Note Marking an input or output port as a local variable does not imply that the code generator uses a local variable in the generated code. If your S-function accesses the inputs and outputs only in its `mdlOutputs` routine, the code generator declares the inputs and outputs as local variables. However, if the inputs and outputs are used elsewhere in the S-function, the code generator includes them in the global block input and output structure.

The reusability setting indicates if the memory associated with an input or output port can be overwritten. To reuse input and output port memory:

1. Indicate the ports are reusable using either the `SS_REUSABLE_AND_LOCAL` or `SS_REUSABLE_AND_GLOBAL` option in the `ssSetInputPortOptimOpts` and `ssSetOutputPortOptimOpts` macros
2. Indicate the input port memory is overwritable using `ssSetInputPortOverWritable`
3. If your S-function has multiple input and output ports, use `ssSetOutputPortOverwritesInputPort` to indicate which output and input ports share memory

The following example shows how different scope and reusability settings affect the generated code. The following model contains an S-function block pointing to the C

MEX S-function *matlabroot*`/toolbox/simulink/simdemos/simfeatures/src/sfun_directlook.c`, which models a direct 1-D lookup table.



The S-function's `mdlInitializeSizes` method declares the input port as reusable, local, and overwritable and the output port as reusable and local, as follows:

```
static void mdlInitializeSizes(SimStruct *S)
{
/* snip */
    ssSetInputPortOptimOpts(S, 0, SS_REUSABLE_AND_LOCAL);
    ssSetInputPortOverWritable(S, 0, TRUE);

/* snip */
    ssSetOutputPortOptimOpts(S, 0, SS_REUSABLE_AND_LOCAL);

/* snip */
}
```

The generated code for this model stores the input and output signals in a single local variable `rtb_SFunction`, as shown in the following output function:

```
static void sl_directlook_output(int_T tid)
{
  /* local block i/o variables */
  real_T rtb_SFunction[2];

  /* Sin: '<Root>/Sine Wave' */
  rtb_SFunction[0] = sin(((real_T)sl_directlook_DWork.counter[0] +
    sl_directlook_P.SineWave_Offset) * 2.0 * 3.1415926535897931E+000 /
    sl_directlook_P.SineWave_NumSamp) * sl_directlook_P.SineWave_Amp[0] +
    sl_directlook_P.SineWave_Bias;
  rtb_SFunction[1] = sin(((real_T)sl_directlook_DWork.counter[1] +
    sl_directlook_P.SineWave_Offset) * 2.0 * 3.1415926535897931E+000 /
    sl_directlook_P.SineWave_NumSamp) * sl_directlook_P.SineWave_Amp[1] +
    sl_directlook_P.SineWave_Bias;

  /* S-Function Block: <Root>/S-Function */
  {
    const real_T *xData = &sl_directlook_P.SFunction_XData[0];
    const real_T *yData = &sl_directlook_P.SFunction_YData [0];
    real_T spacing = xData[1] - xData[0];
    if (rtb_SFunction[0] <= xData[0] ) {
      rtb_SFunction[0] = yData[0];
    } else if (rtb_SFunction[0] >= yData[20] ) {
```

```
      rtb_SFunction[0] = yData[20];
    } else {
      int_T idx = (int_T)( ( rtb_SFunction[0] - xData[0] ) / spacing );
      rtb_SFunction[0] = yData[idx];
    }

    if (rtb_SFunction[1] <= xData[0] ) {
      rtb_SFunction[1] = yData[0];
    } else if (rtb_SFunction[1] >= yData[20] ) {
      rtb_SFunction[1] = yData[20];
    } else {
      int_T idx = (int_T)( ( rtb_SFunction[1] - xData[0] ) / spacing );
      rtb_SFunction[1] = yData[idx];
    }
  }

  /* Outport: '<Root>/Out1' */
  sl_directlook_Y.Out1[0] = rtb_SFunction[0];
  sl_directlook_Y.Out1[1] = rtb_SFunction[1];
  UNUSED_PARAMETER(tid);
}
```

The following table shows variations of the code generated for this model when using the generic real-time target (GRT). Each row explains a different setting for the scope and reusability of the S-function's input and output ports.

Scope and reusability	S-function `mdlInitializeSizes` code	Generated code

Inputs: Local, reusable, overwritable
 	
Outputs: Local, reusable
	`ssSetInputPortOptimOpts(S, 0, SS_REUSABLE_AND_LOCAL);`
	
	`ssSetInputPortOverWritable(S, 0, TRUE);`
	
	`ssSetOutputPortOptimOpts(S, 0, SS_REUSABLE_AND_LOCAL);`
		The *`model`*`.c` file declares a local variable in the output function.
		
		`/* local block i/o variables */`
		`  real_T rtb_SFunction[2];`

Inputs: Global, reusable, overwritable
 	
Outputs: Global, reusable
	`ssSetInputPortOptimOpts(S, 0, SS_REUSABLE_AND_GLOBAL);`
	
	`ssSetInputPortOverWritable(S, 0, TRUE);`
	
	`ssSetOutputPortOptimOpts(S, 0, SS_REUSABLE_AND_GLOBAL);`
		The *`model`*`.h` file defines a block signals structure with a single element to store the S-function's input and output.
		
		`/* Block signals (auto storage) */`
		`typedef struct {`
		`  real_T SFunction[2];`
		`} BlockIO_sl_directlook;`
		The *`model`*`.c` file uses this element of the structure in calculations of the S-function's input and output signals.
		
		`/* Sin: '<Root>/Sine Wave' */`

Scope and reusability	S-function `mdlInitializeSizes` code	Generated code

		`sl_directlook_B.SFunction[0] = sin ...`
		`/* snip */`
		`/*S-Function Block:<Root>/S-Function*/`
		`{`
		`const real_T *xData =`
		`  &sl_directlook_P.SFunction_XData[0]`

Inputs: Local, not reusable
 	
Outputs: Local, not reusable
	`ssSetInputPortOptimOpts(S, 0, SS_NOT_REUSABLE_AND_LOCAL);`
	
	`ssSetInputPortOverWritable(S, 0, FALSE);`
	
	`ssSetOutputPortOptimOpts(S, 0, SS_NOT_REUSABLE_AND_LOCAL);`
		The *`model`*`.c` file declares local variables for the S-function's input and output in the output function
		
		`/* local block i/o variables */`
		`  real_T rtb_SineWave[2];`
		`  real_T rtb_SFunction[2];`

Inputs: Global, not reusable
 	
Outputs: Global, not reusable
	`ssSetInputPortOptimOpts(S, 0, SS_NOT_REUSABLE_AND_GLOBAL);`
	
	`ssSetInputPortOverWritable(S, 0, FALSE);`
	
	`ssSetOutputPortOptimOpts(S, 0, SS_NOT_REUSABLE_AND_GLOBAL);`
		The *`model`*`.h` file defines a block signal structure with individual elements to store the S-function's input and output.
		
		`/* Block signals (auto storage) */`
		`typedef struct {`
		`  real_T SineWave[2];`
		`  real_T SFunction[2];`
		`} BlockIO_sl_directlook;`
		The *`model`*`.c` file uses the different elements in this structure when calculating the S-function's input and output.
		
		`/* Sin: '<Root>/Sine Wave' */`
		`sl_directlook_B.SineWave[0] = sin ...`
		`/* snip */`
		`/*S-Function Block:<Root>/S-Function*/`
		`{`
		`const real_T *xData =`
		`  &sl_directlook_P.SFunction_XData[0]`

S-Functions That Specify Sample Time Inheritance Rules

For the Simulink engine to determine whether a model can inherit a sample time, the S-functions in the model need to specify how they use sample times. You can specify this information by calling the macro `ssSetModelReferenceSampleTimeInheritanceRule` from `mdlInitializeSizes` or `mdlSetWorkWidths`. To use this macro:

1 Check whether the S-function calls any of the following macros:

- `ssGetSampleTime`
- `ssGetInputPortSampleTime`
- `ssGetOutputPortSampleTime`
- `ssGetInputPortOffsetTime`
- `ssGetOutputPortOffsetTime`
- `ssGetSampleTimePtr`
- `ssGetInputPortSampleTimeIndex`
- `ssGetOutputPortSampleTimeIndex`
- `ssGetSampleTimeTaskID`
- `ssGetSampleTimeTaskIDPtr`

2 Check for the following in your S-function TLC code:

- `LibBlockSampleTime`
- `CompiledModel.SampleTime`
- `LibBlockInputSignalSampleTime`
- `LibBlockInputSignalOffsetTime`
- `LibBlockOutputSignalSampleTime`
- `LibBlockOutputSignalOffsetTime`

3 Depending on your search results, use `ssSetModelReferenceSampleTimeInheritanceRule` as indicated in the following table.

If...	Use...
None of the macros or functions are present, the S-function does not preclude the model from inheriting a sample time.	`ssSetModelReferenceSampleTimeInheritanceRule (S, USE_DEFAULT_FOR_DISCRETE_INHERITANCE)`
Any of the macros or functions are used for • Throwing errors if sample time is inherited, continuous, or constant	`ssSetModelReferenceSampleTimeInheritanceRule... (S,USE_DEFAULT_FOR_DISCRETE_INHERITANCE)`

If...	Use...
• Checking `ssIsSampleHit` • Checking whether sample time is inherited in either `mdlSetInputPortSampleTime` or `mdlSetOutputPortSampleTim` before setting	
The S-function uses its sample time for computing parameters, outputs, and so on	`ssSetModelReferenceSampleTimeInheritanceRule (S, DISALLOW_SAMPLE_TIME_INHERITANCE)`

Note If an S-function does not set the `ssSetModelReferenceSampleTimeInheritanceRule` macro, by default the Simulink engine assumes that the S-function does not preclude the model containing that S-function from inheriting a sample time. However, the engine issues a warning indicating that the model includes S-functions for which this macro is not set.

You can use settings on the **All Parameters** pane of the Configuration Parameters dialog box or Model Explorer to control how the Simulink engine responds when it encounters S-functions that have unspecified sample time inheritance rules. Toggle the **Unspecified inheritability of sample time** diagnostic to `none`, `warning`, or `error`. The default is `warning`.

S-Functions That Support Code Reuse

You can reuse the generated code for identical subsystems that occur in multiple instances within a model and across referenced models. For more information about code generation of subsystems for code reuse, see "Code Generation of Subsystems" (Simulink Coder). If you want your S-function to support code reuse for a subsystem, the S-function must meet the following requirements:

- The S-function must be inlined.
- Code generated from the S-function must not use static variables.
- The S-function must initialize its pointer work vector in `mdlStart` and not before.
- The S-function must not be a sink that logs data to the workspace.

- The S-function must register its parameters as run-time parameters in `mdlSetWorkWidths`. (It must not use `ssWriteRTWParameters` in its `mdlRTW` function for this purpose.)
- The S-function must not be a device driver.

In addition to meeting the preceding requirements, your S-function must set the `SS_OPTION_WORKS_WITH_CODE_REUSE` flag (see the description of `ssSetOptions` in the Simulink Writing S-Function documentation). This flag indicates that your S-function meets the requirements for subsystem code reuse.

S-Functions for Multirate Multitasking Environments

About S-Functions for Multirate Multitasking Environments

S-functions can be used in models with multiple sample rates and deployed in multitasking target environments. Likewise, S-functions themselves can have multiple rates at which they operate. The code generator produces code for multirate multitasking models using an approach called *rate grouping*. In code generated for ERT-based targets, rate grouping generates separate *`model`*`_step` functions for the base rate task and each subrate task in the model. Although rate grouping is a code generation feature found in ERT targets only, your S-functions can use it in other contexts when you code them as explained below.

Rate Grouping Support in S-Functions

To take advantage of rate grouping, you must inline your multirate S-functions if you have not done so. You need to follow certain Target Language Compiler protocols to exploit rate grouping. Coding TLC to exploit rate grouping does not prevent your inlined S-functions from functioning properly in GRT. Likewise, your inlined S-functions will still generate valid ERT code even if you do not make them rate-grouping-compliant. If you do so, however, they will generate more efficient code for multirate models.

For instructions and examples of Target Language Compiler code illustrating how to create and upgrade S-functions to generate rate-grouping-compliant code, see "Rate Grouping Compliance and Compatibility Issues" on page 49-17.

For each multirate S-function that is not rate grouping-compliant, the code generator issues the following warning when you build:

```
Warning:  Simulink Coder: Code of output function for multirate block
'<Root>/S-Function' is guarded by sample hit checks rather than being rate
grouped. This will generate the same code for all rates used by the block,
possibly generating dead code. To avoid dead code, you must update the TLC
file for the block.
```

You will also find a comment such as the following in code generated for each noncompliant S-function:

```
/* Because the output function of multirate block
   <Root>/S-Function is not rate grouped,
   the following code might contain unreachable blocks of code.
   To avoid this, you must update your block TLC file. */
```

The words “update function” are substituted for “output function” in these warnings.

Create Multitasking, Multirate, Port-Based Sample Time S-Functions

The following instructions show how to support both data determinism and data integrity in multirate S-functions. They do not cover cases where there is no determinism nor integrity. Support for frame-based processing does not affect the requirements.

Note The slow rates must be multiples of the fastest rate. The instructions do not apply when two rates being interfaced are not multiples or when the rates are not periodic.

Rules for Properly Handling Fast-to-Slow Transitions

The rules that multirate S-functions should observe for inputs are

- The input should only be read at the rate that is associated with the input port sample time.
- Generally, the input data is written to `DWork`, and the `DWork` can then be accessed at the slower (downstream) rate.

The input can be read at every sample hit of the input rate and written into `DWork` memory, but this `DWork` memory cannot then be directly accessed by the slower rate. `DWork` memory that will be read by the slow rate must only be written by the fast rate when there is a *special sample hit*. A special sample hit occurs when both this input port rate and rate to which it is interfacing have a hit. Depending on their requirements and design, algorithms can process the data in several locations.

The rules that multirate S-functions should observe for outputs are

- The output should not be written by a rate other than the rate assigned to the output port, except in the optimized case described below.
- The output should always be written when the sample rate of the output port has a hit.

If these conditions are met, the S-Function block can specify that the input port and output port can both be made local and reusable.

You can include an optimization when little or no processing needs to be done on the data. In such cases, the input rate code can directly write to the output (instead of by using DWork) when there is a special sample hit. If you do this, however, you must declare the outport port to be *global* and *not reusable*. This optimization results in one less `memcpy` but does introduce nonuniform processing requirements on the faster rate.

Whether you use this optimization or not, the most recent input data, as seen by the slower rate, is the value when both the faster and slower rate had their hits (and possible earlier input data as well, depending on the algorithm). Subsequent steps by the faster rate and the associated input data updates are not seen by the slower rate until the next hit for the slow rate occurs.

Pseudocode Examples of Fast-to-Slow Rate Transition

The pseudocode below abstracts how you should write your C MEX code to handle fast-to-slow transitions, illustrating with an input rate of 0.1 second driving an output rate of one second. A similar approach can be taken when inlining the code. The block has following characteristics:

- File: `sfun_multirate_zoh.c`, Equation: `y = u(tslow)`
- Input: local and reusable
- Output: local and reusable
- DirectFeedthrough: yes

```
OutputFcn
if (ssIsSampleHit(".1")) {
    if (ssIsSepcialSampleHit("1")) {
        DWork = u;
    }
}
if (ssIsSampleHit("1")) {
    y = DWork;
}
```

An alternative, slightly optimized approach for simple algorithms:

- Input: local and reusable
- Output: global and not reusable because it needs to persist between special sample hits
- DirectFeedthrough: yes

```
OutputFcn
if (ssIsSampleHit(".1")) {
    if (ssIsSpecialSampleHit("1")) {
        y = u;
    }
}
```

Example adding a simple algorithm:

- File: `sfun_multirate_avg.c`; Equation: `y = average(u)`
- Input: local and reusable
- Output: local and reusable
- DirectFeedthrough: yes

  (Assume `DWork[0:10]` and `DWork[mycounter]` are initialized to zero)

```
OutputFcn
if (ssIsSampleHit(".1")) {
    /* In general, processing on 'u' could be done here,
        it runs on every hit of the fast rate. */
    DWork[DWork[mycounter]++] = u;
    if (ssIsSpecialSampleHit("1")) {
    /* In general, processing on DWork[0:10] can be done
       here, but it does cause the faster rate to have
       nonuniform processing requirements (every 10th hit,
       more code needs to be run).*/
        DWork[10] = sum(DWork[0:9])/10;
        DWork[mycounter] = 0;
    }
}
if (ssIsSampleHit("1")) {
    /* Processing on DWork[10] can be done here before
       outputing. This code runs on every hit of the
       slower task. */
    y = DWork[10];
}
```

Rules for Properly Handling Slow-to-Fast Transitions

When output rates are faster than input rates, input should only be read at the rate that is associated with the input port sample time, observing the following rules:

- Always read input from the update function.
- Use no special sample hit checks when reading input.
- Write the input to a DWork.
- When there is a special sample hit between the rates, copy the DWork into a second DWork in the output function.
- Write the second DWork to the output at every hit of the output sample rate.

The block can request that the input port be made local but it cannot be set to reusable. The output port can be set to local and reusable.

As in the fast-to-slow transition case, the input should not be read by a rate other than the one assigned to the input port. Similarly, the output should not be written to at a rate other than the rate assigned to the output port.

An optimization can be made when the algorithm being implemented is only required to run at the slow rate. In such cases, you use only one DWork. The input still writes to the DWork in the update function. When there is a special sample hit between the rates, the output function copies the same DWork directly to the output. You must set the output port to be global and not reusable in this case. This optimization results in one less `memcpy` operation per special sample hit.

In either case, the data that the fast rate computations operate on is always delayed, that is, the data is from the previous step of the slow rate code.

Pseudocode Examples of Slow-to-Fast Rate Transition

The pseudocode below abstracts what your S-function needs to do to handle slow-to-fast transitions, illustrating with an input rate of one second driving an output rate of 0.1 second. The block has following characteristics:

- File: `sfun_multirate_delay.c`, Equation: `y = u(tslow-1)`
- Input: Set to local, will be local if output/update are combined (ERT) otherwise will be global. Set to not reusable because input needs to be preserved until the update function runs.
- Output: local and reusable
- DirectFeedthrough: no

```
OutputFcn
if (ssIsSampleHit(".1") {
    if (ssIsSpecialSampleHit("1") {
        DWork[1] = DWork[0];
```

```
    }
    y = DWork[1];
}
UpdateFcn
if (ssIsSampleHit("1")) {
    DWork[0] = u;
}
```

An alternative, optimized approach can be used by some algorithms:

- Input: Set to local, will be local if output/update are combined (ERT) otherwise will be global. Set to not reusable because input needs to be preserved until the update function runs.
- Output: global and not reusable because it needs to persist between special sample hits.
- DirectFeedthrough: no

```
OutputFcn
if (ssIsSampleHit(".1") {
    if (ssIsSpecialSampleHit("1") {
        y = DWork;
    }
}
UpdateFcn
if (ssIsSampleHit("1")) {
    DWork = u;
}
```

Example adding a simple algorithm:

- File: `sfun_multirate_modulate.c`, Equation: `y = sin(tfast) + u(tslow-1)`
- Input: Set to local, will be local if output/update are combined (an ERT feature) otherwise will be global. Set to not reusable because input needs to be preserved until the update function runs.
- Output: local and reusable
- DirectFeedthrough: no

```
OutputFcn
if (ssIsSampleHit(".1") {
    if (ssIsSpecialSampleHit("1") {
    /* Processing not likely to be done here. It causes
     * the faster rate to have nonuniform processing
     * requirements (every 10th hit, more code needs to
     * be run).*/
        DWork[1] = DWork[0];
    }
    /* Processing done at fast rate */
    y = sin(ssGetTaskTime(".1")) + DWork[1];
```

```
}
UpdateFcn
if (ssIsSampleHit("1")) {
    /* Processing on 'u' can be done here. There is a delay of
       one slow rate period before the fast rate sees it.*/
    DWork[0] = u;}
```

See Also

`legacy_code`

Related Examples

- "Introduction to the Target Language Compiler" (Simulink Coder)
- "Inlining S-Functions" (Simulink Coder)
- "Import Calls to External Code into Generated Code with Legacy Code Tool" on page 11-7

Guidelines and Standards for Embedded Coder

Support for Standards and Guidelines

If your application has mission-critical development and certification goals, your models or subsystems and the code generated for them might need to comply with one or more of the standards and guidelines listed in the following table.

Standard or Guidelines	Organization	For More Information, See...
Guidelines: Use of MATLAB, Simulink, and Stateflow software for control algorithm modeling – MathWorks Automotive Advisory Board (MAAB) Guidelines	MAAB	• Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow Software: MathWorks Automotive Advisory Board (MAAB) Guidelines • Develop Models and Code That Comply with "MAAB Guidelines" on page 12-4
Guidelines: Use of the C Language in Critical Systems (MISRA C®[a])	Motor Industry Software Reliability Association (MISRA)	• MISRA C website • Technical Solution 1-1IFP0W on the MathWorks website • Develop Models and Code That Comply with "MISRA C Guidelines" on page 12-5
Standard: AUTomotive Open System ARchitecture (AUTOSAR)	AUTOSAR Development Partnership	• Publications and specifications available from the AUTOSAR website • AUTOSAR Support from Embedded Coder on the MathWorks website • "AUTOSAR Standard" • Embedded Coder "AUTOSAR" documentation
Standard: IEC 61508, Functional safety of electrical/ electronic/programmable electronic safety-related systems	International Electrotechnical Commission	• IEC functional safety zone website • IEC 61508 Support in MATLAB and Simulink

Standard or Guidelines	Organization	For More Information, See...
		• Develop Models and Code That Comply with "IEC 61508 Standard" on page 12-7
Standard: IEC 62304, Medical device software - Software life cycle processes	International Electrotechnical Commission	• Develop Models and Code That Comply with "IEC 62304 Standard" on page 12-12
Standard: ISO 26262, Road Vehicles - Functional Safety	International Organization for Standardization	• ISO 26262 Support in MATLAB and Simulink • Develop Models and Code That Comply with "ISO 26262 Standard" on page 12-13
Standard: EN 50128, Railway applications — Software for railway control and protection systems	European Committee for Electrotechnical Standardization	• Develop Models and Code That Comply with "EN 50128 Standard" on page 12-15
Standard: DO-178C, Software Considerations in Airborne Systems and Equipment Certification	Radio Technical Commission for Aeronautics (RTCA)	• Develop Models and Code That Comply with "DO-178C Standard" on page 12-17

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

MAAB Guidelines

The MathWorks Automotive Advisory Board (MAAB) involves major automotive OEMs and suppliers in the process of evolving MathWorks controls, simulation, and code generation products, including Simulink, Stateflow, and Simulink Coder. An important result of the MAAB has been the "MAAB Control Algorithm Modeling" (Simulink) guidelines.

If you have a Simulink Verification and Validation product license, you can check that your Simulink model or subsystem, and the code that you generate from it, complies with MAAB guidelines. To check your model or subsystem, open the Simulink Model Advisor (Simulink). Navigate to **By Product > Simulink Verification and Validation > Modeling Standards > MathWorks Automotive Advisory Board Checks** and run the MathWorks Automotive Advisory Board checks (Simulink Verification and Validation).

For more information on using the Model Advisor, see "Run Model Checks" (Simulink).

MISRA C Guidelines

The Motor Industry Software Reliability Association (MISRA[2]) has established "Guidelines for the Use of the C Language in Critical Systems" (MISRA C).

For information about MISRA C, see www.misra.org.uk.

In 1998, MIRA Ltd. published MISRA C (MISRA C:1998) to provide a restricted subset of a standardized, structured language that met Safety Integrity Level (SIL) 2 and higher. A major update based on feedback was published in 2004 (MISRA C:2004), followed by a minor update in 2007 known as Technical Corrigendum (TC1).

In 2007, MIRA Ltd. published the MISRA AC AGC standard, "MISRA AC AGC: Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation." MISRA AC AGC does not change MISRA C:2004 rules, rather it modifies the adherence recommendation.

In 2013, MIRA Ltd. published the MISRA C:2012 standard, "Guidelines for the use of the C language in critical systems." MISRA C:2012 provides improvements based on user feedback and includes guidance on automatic code generation.

Embedded Coder and Simulink offer capabilities to minimize the potential for MISRA C rule violations.

To configure a model or subsystem so that the code generator is most likely to produce MISRA C:2012 compliant code, use the Code Generation Advisor. For more information, see "Configure Model for Code Generation Objectives by Using Code Generation Advisor" on page 29-2.

The Model Advisor (Simulink) also checks that you developed your model or subsystem to increase the likelihood of generating MISRA C:2012 compliant code. To check your model or subsystem:

1. Open the Model Advisor.
2. Navigate to **By Task > Modeling Guidelines for MISRA C:2012**.
3. Run the checks in the folder.

For more information about using the Model Advisor, see "Run Model Checks" (Simulink).

2. MISRA and MISRA C are registered trademarks of MIRA Ltd., held on behalf of the MISRA Consortium.

For information about using Embedded Coder software within MISRA C guidelines, see Technical Solution 1-1IFP0W on the MathWorks website.

IEC 61508 Standard

In this section...

Apply Simulink and Embedded Coder to the IEC 61508 Standard

Applying Model-Based Design to a safety-critical system requires extra consideration and rigor so that the system adheres to defined safety standards. IEC 61508, Functional safety of electrical/electronic/programmable electronic safety related systems, is such a standard. Because the standard was published when most software was coded by hand, the standard needs to be mapped to Model-Based Design technologies. For further information about MathWorks support for IEC 61508, see IEC 61508 Support in MATLAB and Simulink.

MathWorks provides an IEC Certification Kit product that you can use to certify MathWorks code generation and verification tools for projects based on the IEC 61508 standard. For more information, see http://www.mathworks.com/products/iec-61508/.

Check for IEC 61508 Standard Compliance Using the Model Advisor

If you have a Simulink Verification and Validation product license, you can check that your Simulink model or subsystem and the code that you generate from it complies with selected aspects of the IEC 61508 standard by running the Simulink Model Advisor (Simulink). Navigate to **By Task > Modeling Standards for IEC 61508** and run the "IEC 61508, IEC 62304, ISO 26262, and EN 50128 Checks" (Simulink Verification and Validation).

For more information on using the Model Advisor, see "Run Model Checks" (Simulink).

Validate Traceability

Typically, applications that require certification require some level of traceability between requirements, models, and corresponding code.

To...	Use...
Associate requirements documents with objects in Simulink models	The “Requirements Traceability” (Simulink Verification and Validation) that is available if you have a Simulink Verification and Validation license.
Trace model blocks and subsystems to generated code	The **Model-to-code** traceability option on page 61-8 when generating an HTML report during the code generation or build process.
Trace generated code to model blocks and subsystems	The **Code-to-model** traceability option on page 61-6 when generating an HTML report during the code generation or build process.

Develop a Model that Complies with the IEC 61508 Standard

This example shows how to use Model Advisor checks for the IEC 61508 standard to develop a model and code that comply with the standard.

The IEC 61508 checks identify issues with a model that impede deployment in safety-related applications or limit traceability.

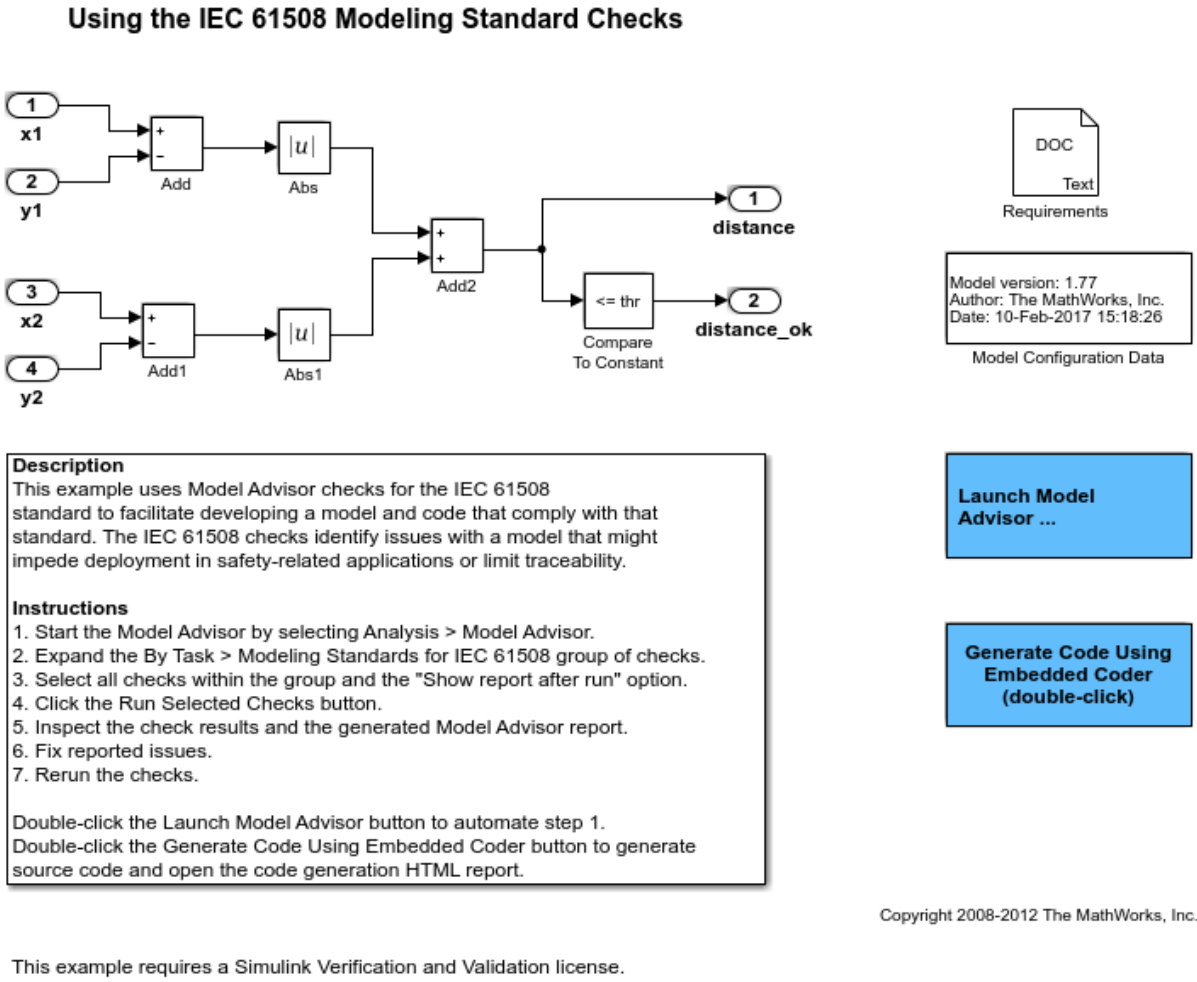
Understanding the Model

According to the functional requirements, a model shall be created that checks whether the 1-norm distance between points `(x1,x2)` and `(y1,y2)` is less than or equal to a given threshold `thr`. For two points `(x1,x2)` and `(y1,y2)`, the 1-norm distance is given as:

$$\sum_{i=1}^{2} |x_i - y_i|$$

The rtwdemo_iec61508 model implements the preceding requirement. Open and get familiar with the model.

```
model='rtwdemo_iec61508';
open_system(model)
```



Apply the IEC 61508 Modeling Standard Checks

To deploy the model in a safety-related software component that must comply with the IEC 61508 safety standard, check the model for issues that might impede deployment in such an environment or limit traceability between the model and generated source code.

To identify possible compliance issues with the model:

1. Start the Model Advisor by selecting **Analysis > Analysis > Model Advisor** or by entering modeladvisor('rtwdemo_IEC61508') at the MATLAB command line.
2. In the **Task Hierarchy**, expand **By Task > Modeling Standards for IEC 61508**.
3. Select the checks within the group.
4. Select **Show report after run** to generate an HTML report that shows the check results.
5. Click **Run Selected Checks**. Model Advisor processes the IEC 61508 checks and displays the results.

To review the check results and make changes:

1. Review the **Summary** in the **Report** section of the right pane.
2. In the **Task Hierarchy**, select a check that did not pass. Review the results that appear in the right pane for that check. For more information on the check and on how to resolve reported issues, with the check selected, click **Help**.
3. Click the **Generate Code Using Embedded Coder** button in the model to inspect the generated code and the traceability report.
4. Resolve the reported issues and rerun the checks.
5. Review the generated HTML report of the check results by clicking the link in the **Report** box.
6. Print the generated HTML report. You can use the report as evidence in the IEC 61508 compliance example process.

See Also

- For descriptions of the IEC 61508 checks, see IEC 61508, IEC 62304, ISO 26262, and EN 50128 Checks in the Simulink Verification and Validation documentation.
- For more information on using Model Advisor, see Run Model Checks in the Simulink documentation.

IEC 62304 Standard

Apply Simulink and Embedded Coder to the IEC 62304 Standard

Applying Model-Based Design to a safety-critical system requires extra consideration and rigor so that the system adheres to defined safety standards. Standard: IEC 62304, Medical device software - Software life cycle processes, is such a standard.

MathWorks provides an IEC Certification Kit product that you can use to certify MathWorks code generation and verification tools for projects based on the IEC 62304 standard. For more information, see http://www.mathworks.com/products/iec-61508/.

Check for IEC 62304 Standard Compliance Using the Model Advisor

If you have a Simulink Verification and Validation product license, you can check that your Simulink model or subsystem and the code that you generate from it complies with selected aspects of the IEC 62304 standard by running the Simulink Model Advisor (Simulink). Navigate to **By Task > Modeling Standards for IEC 62304** and run the "IEC 61508, IEC 62304, ISO 26262, and EN 50128 Checks" (Simulink Verification and Validation).

For more information on using the Model Advisor, see "Run Model Checks" (Simulink).

ISO 26262 Standard

In this section...

Apply Simulink and Embedded Coder to the ISO 26262 Standard

Applying Model-Based Design to a safety-critical system requires extra consideration and rigor so that the system adheres to defined functional safety standards. ISO 26262, Road Vehicles - Functional Safety, is such a standard. For further information about MathWorks support for ISO 26262, see ISO 26262 Support in MATLAB and Simulink.

MathWorks provides an IEC Certification Kit product that you can use to qualify MathWorks code generation and verification tools for projects based on the ISO 26262 standard. For more information, see http://www.mathworks.com/products/iso–26262/.

Check for ISO 26262 Standard Compliance Using the Model Advisor

If you have a Simulink Verification and Validation product license, you can check that your Simulink model or subsystem and the code that you generate from it complies with selected aspects of the ISO 26262 standard by running the Simulink Model Advisor (Simulink). Navigate to **By Task > Modeling Standards for ISO 26262** and run the "IEC 61508, IEC 62304, ISO 26262, and EN 50128 Checks" (Simulink Verification and Validation).

For more information on using the Model Advisor, see "Run Model Checks" (Simulink).

Validate Traceability

Typically, applications that require certification require some level of traceability between requirements, models, and corresponding code.

To...	Use...
Associate requirements documents with objects in Simulink models	The "Requirements Traceability" (Simulink Verification and Validation) that is available if you have a Simulink Verification and Validation license.

To...	Use...
Trace model blocks and subsystems to generated code	The **Model-to-code** traceability option on page 61-8 when generating an HTML report during the code generation or build process.
Trace generated code to model blocks and subsystems	The **Code-to-model** traceability option on page 61-6 when generating an HTML report during the code generation or build process.

EN 50128 Standard

In this section...

Apply Simulink and Embedded Coder to the EN 50128 Standard

Applying Model-Based Design to a safety-critical system requires extra consideration and rigor so that the system adheres to defined safety standards. EN 50128, Railway applications — Software for railway control and protection systems, is such a standard.

MathWorks provides an IEC Certification Kit product that you can use to certify MathWorks code generation and verification tools for projects based on the EN 50128 standard. For more information, see http://www.mathworks.com/products/iec-61508/.

Check for EN 50128 Standard Compliance Using the Model Advisor

If you have a Simulink Verification and Validation product license, you can check that your Simulink model or subsystem and the code that you generate from it complies with selected aspects of the EN 50128 standard by running the Simulink Model Advisor (Simulink). Navigate to **By Task > Modeling Standards for EN 50128** and run the “IEC 61508, IEC 62304, ISO 26262, and EN 50128 Checks” (Simulink Verification and Validation).

For more information on using the Model Advisor, see “Run Model Checks” (Simulink).

Validate Traceability

Typically, applications that require certification require some level of traceability between requirements, models, and corresponding code.

To...	Use...
Associate requirements documents with objects in Simulink models	The “Requirements Traceability” (Simulink Verification and Validation) that is available if you have a Simulink Verification and Validation license.

To...	Use...
Trace model blocks and subsystems to generated code	The **Model-to-code** traceability option on page 61-8 when generating an HTML report during the code generation or build process.
Trace generated code to model blocks and subsystems	The **Code-to-model** traceability option on page 61-6 when generating an HTML report during the code generation or build process.

DO-178C Standard

In this section...

Apply Simulink and Embedded Coder to the DO-178C Standard

Applying Model-Based Design to a high-integrity system requires extra consideration and rigor so that the system adheres to defined safety standards. DO-178C Software Considerations in Airborne Systems and Equipment Certification is such a standard. A supplement to DO-178C, DO-331, provides guidance on the use of Model-Based Design technologies. MathWorks provides a DO Qualification Kit product that you can use to qualify MathWorks verification tools for projects based on the DO-178C, DO-331, and related standards. For more information, see http://www.mathworks.com/products/do-178/.

For information about Model-Based Design and MathWorks support of aerospace and defense industry standards, see http://www.mathworks.com/aerospace-defense/ .

Check for Standard Compliance Using the Model Advisor

If you have a Simulink Verification and Validation product license, you can check that your Simulink model or subsystem and the code that you generate from it complies with selected aspects of the DO-178C standard by running the Simulink Model Advisor (Simulink). Navigate to **By Product > Simulink Verification and Validation > Modeling Standards > DO-178C/DO-331 Checks** or **By Task > Modeling Standards for DO-178C/DO-331** and run the DO-178C/DO-331 checks (Simulink Verification and Validation).

For more information on using the Model Advisor, see "Run Model Checks" (Simulink).

Validate Traceability

Typically, applications that require certification require some level of traceability between requirements, models, and corresponding code.

To...	Use...
Associate requirements documents with objects in Simulink models	The “Requirements Traceability” (Simulink Verification and Validation) that is available if you have a Simulink Verification and Validation license.
Trace model blocks and subsystems to generated code	The **Model-to-code** traceability option on page 61-8 when generating an HTML report during the code generation or build process.
Trace generated code to model blocks and subsystems	The **Code-to-model** traceability option on page 61-6 when generating an HTML report during the code generation or build process.

Patterns for C Code in Embedded Coder

Prepare a Model for Code Generation

In this section...

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

Model examples in “Modeling Patterns for C Code” have the following naming conventions:

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

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

Configure a Signal

1 Create a model in Simulink. For more information, see “Model Editing Fundamentals” (Simulink).

2 Right-click a signal line. Select **Properties**. For more information about the Signal Properties dialog box, see "Signal Properties" (Simulink).

3 Enter a signal name for the **Signal name** parameter.

4 On the same Signal Properties dialog box, select the **Code Generation** tab. Use the drop down menu for the **Storage class** parameter to specify a storage class. Examples in this chapter use `ExportedGlobal`.

Note: Alternatively, on the Signal Properties dialog box, select **Signal name must resolve to Simulink signal object**. Then create a signal data object in the base workspace with the same name as the signal. See "Create Data Objects for Code Generation with Data Object Wizard" on page 24-2 for more information on creating data objects in the base workspace. (Examples use `Simulink.Signal` and specify the **Storage class** as `ExportedGlobal`.)

Configure Input and Output Ports

1 In your model,

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

2 Select the **Signal Attributes** tab.

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

Initialize States

1 Double-click a block.

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

3 Specify the **Initial conditions** and **Sample time**. For more information, see "Specify Sample Time" (Simulink).

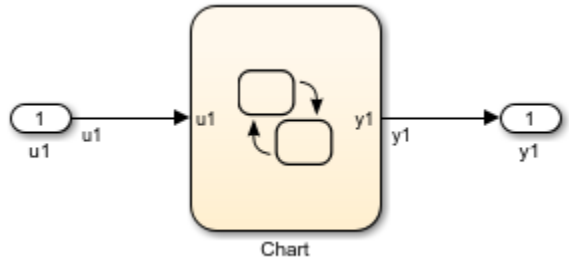
4 Select the **State Attributes** pane. Specify the state name. See "Discrete Block State Naming in Generated Code" (Simulink Coder).

5 You can also use the Data Object Wizard for creating data objects. A part of this process initializes states. See "Create Data Objects for Code Generation with Data Object Wizard" on page 24-2.

Set Up Configuration Parameters for Code Generation

1. Open the Configuration Parameter dialog box by selecting **Simulation** > **Model Configuration parameters**. You can also use the keyboard shortcut `Ctrl+E`.
2. Open the **Solver** pane and select
   - **Solver type**: `Fixed-Step`
   - **Solver**: `discrete (no continuous states)`
3. Open the **Optimization** > **Signals and Parameters** pane, and set **Default parameter behavior** to `Inlined`.
4. Open the **Code Generation** pane, and specify `ert.tlc` as the **System Target File**.
5. Clear **Generate makefile**.
6. Select **Generate code only**.
7. Enable the HTML report generation by opening the **Code Generation** > **Report** pane and selecting **Create code generation report** and **Launch report automatically**. On the **All Parameters** tab, select **Code-to-model**. Enabling the HTML report generation is optional.
8. Click **Apply** and then **OK** to exit.

Set Up an Example Model With a Stateflow Chart



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

1. From the **Stateflow > Chart** library, add a Stateflow chart to your model .
2. Add Inport blocks and Outport blocks according to the example model.

3 Open the **Stateflow Editor** by performing one of the following:

- Double-click the Stateflow chart.
- Press **Ctrl+R**.

4 Select **Chart > Add Inputs & Outputs > Data Input from Simulink** to add the inputs to the chart. A Data dialog box opens for each input.

5 Specify the **Name** (`u1, u2, ...`) and the **Type** (`Inherit: Same as Simulink`) for each input, unless specified differently in the example. Click **OK**.

Click **Apply** and close each dialog box.

6 Select **Chart > Add Inputs & Outputs > Data Output from Simulink** to add the outputs to the chart. A Data dialog opens for each output.

7 Specify the **Name** (`y1, y2, ...`) and **Type** (`Inherit: Same as Simulink`) for each output, unless specified differently in the example. Click **OK**.

8 Click **Apply** and close each dialog box.

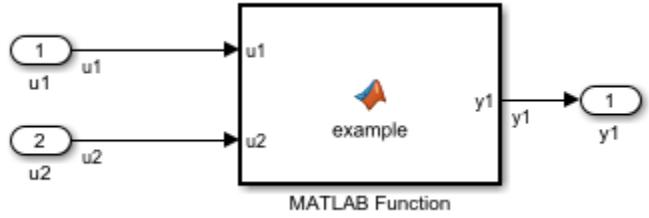
9 In the **Stateflow Editor**, create the Stateflow diagram specific to the example.

10 The inputs and outputs appear on the chart in your model.

11 Connect the Inport and Outport blocks to the Stateflow Chart.

12 Configure the input and output signals; see “Configure a Signal” on page 13-3.

Set Up an Example Model With a MATLAB Function Block



1 Add the number of Inport and Outport blocks according to a C construct example included in this chapter.

2 From the Simulink User-defined Functions library drag a MATLAB Function block into the model.

3. Double-click the block. The MATLAB Function Block Editor opens. Edit the function to implement your application.
4. Click **File > Save** and close the MATLAB Function Block Editor.
5. Connect the Inport and Outport blocks to the MATLAB Function block. See "Configure a Signal" on page 13-3.
6. Save your model.

Definition, Initialization, and Declaration of Parameter Data

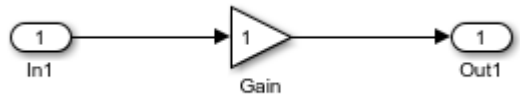
This example shows how to export the definition, initialization, and declaration of a global variable that the generated code uses as a parameter.

C Construct

```
int32 myParam = 3;

extern int32 myParam;
```

Procedure

1. Create the `ex_defn_decl` model by using a Gain block.



2. In the Gain block dialog box, set **Gain** to `myParam`. Click **Apply**.
3. Click the button ⋮ next to the parameter value. Select **Create Variable**.
4. In the Create New Data dialog box, set **Value** to `Simulink.Parameter(3)`. Click **Create**.

   A `Simulink.Parameter` object, `myParam`, appears in the base workspace. The Gain block uses the object to set the value of the **Gain** parameter, in this case, 3.
5. In the Simulink.Parameter property dialog box, set **Data type** to `int32`.
6. Set **Storage class** to `ExportToFile`.
7. Set **HeaderFile** to `myDecls.h`.
8. Set **DefinitionFile** to `myDefns.c`. Click **OK**.
9. Generate code from the model.

Results

The generated header file `myDecls.h` declares the global variable `myParam` by using the `extern` keyword.

```
/* Declaration for custom storage class: ExportToFile */
extern int32_T myParam;
```

The generated source file `myDefns.c` defines and initializes `myParam`.

```
/* Definition for custom storage class: ExportToFile */
int32_T myParam = 3;
```

Related Examples

- "Block Parameter Representation in the Generated Code" on page 19-47
- "Exchange and Reuse Parameter Data Between Generated Code and Existing Code" on page 23-11
- "Manage Placement of Data Definitions and Declarations" on page 36-100

Definition and Declaration of Signal Data

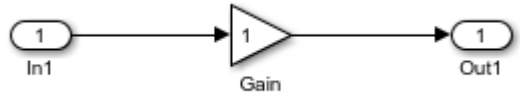
This example shows how to export the definition and declaration of a global variable that the generated code uses as a signal.

C Construct

```
float mySig;

extern float mySig;
```

Procedure

1. Create the `ex_defn_decl` model by using a Gain block.



2. In the model, select **View** > **Model Data**.
3. In the Model Data Editor, view the **Inports/Outports** tab.
4. From the **Change View** drop-down list, select `Design`.
5. In the model, select the Inport block.
6. In the Model Data Editor, for the Inport block, set **Signal Name** to `mySig`.
7. Set **Data Type** to `single`.
8. From the **Change View** drop-down list, select `Code`.
9. For the Inport block, set **Storage Class** to `ExportToFile`.
10. Set **Header File** to `myDecls.h`.
11. Set **Definition File** to `myDefns.c`.
12. Generate code from the model.

Results

The generated header file `myDecls.h` declares the global variable `mySig` by using the `extern` keyword.

```
/* Declaration for custom storage class: ExportToFile */
extern real32_T mySig;
```

The generated source file `myDefns.c` defines the variable `mySig`.

```
/* Definition for custom storage class: ExportToFile */
real32_T mySig;
```

Related Examples

- “Signal Representation in Generated Code” on page 19-112
- “Control Signals and States in Code by Applying Storage Classes” on page 19-123

Data Type Conversion

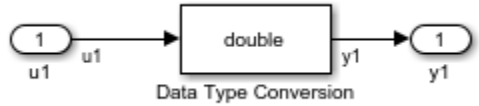
C Construct

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

Modeling Patterns

Modeling Pattern for Data Type Conversion — Simulink Block

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



ex_data_type_SL

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

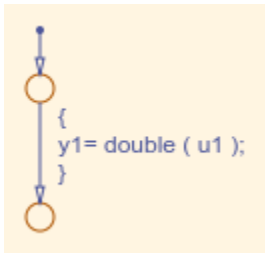
The generated code appears in `ex_data_type_SL.c`, as follows:

```
int32_T u1;
real_T y1;
```

```
void ex_data_type_SL_step(void)
{
 y1 = (real_T)u1;
}
```

The code generator type definition for `double` is `real_T`.

Modeling Pattern for Data Type Conversion — Stateflow Chart



Stateflow Chart Type Conversion

Procedure

1. Follow the steps for “Set Up an Example Model With a Stateflow Chart” on page 13-5 . This example contains one Inport block and one Outport block.
2. Name the example model `ex_data_type_SF`.
3. Double-click the Inport block and select the **Signal Attributes** tab. Specify the **Data Type** as `int32` from the drop down menu.
4. Double-click the Outport block and select the **Signal Attributes** tab. Specify the **Data Type** as `Inherit: auto` from the drop down menu.
5. Press **Ctrl+B** to build the model and generate code.

Results

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

```
int32_T u1;
real_T y1;
RT_MODEL_ex_data_type_SF ex_data_type_SF_M_;
RT_MODEL_ex_data_type_SF *const ex_data_type_SF_M = &ex_data_type_SF_M_;
void ex_data_type_SF_step(void)
{
  y1 = u1;
}
```

Modeling Pattern for Data Type Conversion — MATLAB Function Block

Procedure

1. Follow the steps for "Set Up an Example Model With a MATLAB Function Block" on page 13-6 . This example model contains one Inport block and one Outport block.
2. Name the model `ex_data_type_ML_Func`.
3. In the MATLAB Function Block Editor enter the function, as follows:

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

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

Results

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

```
real32_T u1;
real_T y1;

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

Other Type Conversions in Modeling

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

See Also

Data Type Conversion

Type Qualifiers

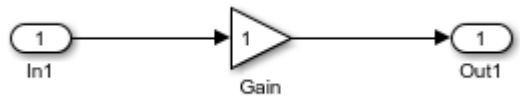
This example shows how to apply the `const` and `volatile` keywords to a global variable that represents parameter data.

C Construct

```
const volatile double myParam = 9.8;
```

Procedure

1. Create the `ex_const_volatile` model by using a Gain block.



2. In the Gain block dialog box, set **Gain** to `myParam`. Click **Apply**.
3. Click the button ⋮ next to the parameter value. Select **Create Variable**.
4. In the Create New Data dialog box, set **Value** to `Simulink.Parameter(9.8)`. Click **Create**.

   A `Simulink.Parameter` object, `myParam`, appears in the base workspace. The Gain block uses the object to set the value of the **Gain** parameter, in this case, 9.8.
5. In the Simulink.Parameter property dialog box, set **Storage class** to `ConstVolatile`. Click **OK**.

   Alternatively, to apply only one of the keywords, you can use the storage classes `Const` or `Volatile`.
6. Generate code from the model.

Results

The generated source file `ex_const_volatile.c` defines `myParam` by using the `const` and `volatile` keywords.

```
/* Definition for custom storage class: ConstVolatile */
```

```
const volatile real_T myParam = 9.8;
```

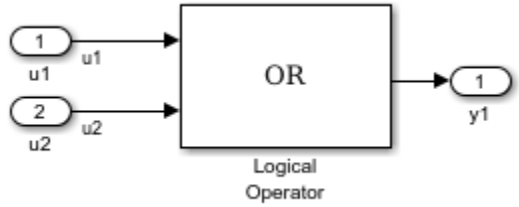
Related Examples

- “Create Tunable Calibration Parameter in the Generated Code” on page 19-60
- “Control Data Representation by Applying Custom Storage Classes” on page 23-58

Relational and Logical Operators

Modeling Patterns for Relational and Logical Operators

Modeling Pattern for Relational or Logical Operators — Simulink Blocks



ex_logical_SL

Procedure

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

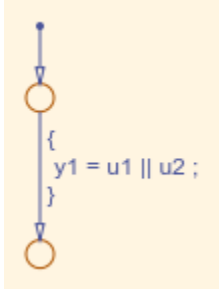
Note: You can use the above procedure to implement relational operators by replacing the Logical Operator block with a Relational Operator block.

Results

Code implementing the logical operator `OR` is in the `ex_logical_SL_step` function in `ex_logical_SL.c`:

```
/* Exported block signals */
   boolean_T u1;                         /* '<Root>/u1' */
   boolean_T u2;                         /* '<Root>/u2' */
   boolean_T y1;                         /* '<Root>/Logical Operator'*/

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

Modeling Pattern for Relational and Logical Operators – Stateflow Chart



ex_logical_SF/Logical Operator Stateflow Chart

Procedure

1. Follow the steps for "Set Up an Example Model With a Stateflow Chart" on page 13-5. This example model contains two Inport blocks and one Outport block.
2. Name the example model `ex_logical_SF`.
3. In the **Stateflow Editor**, specify the **Data Type** for `y1` as `Boolean`.
4. In the **Stateflow Editor**, create the Stateflow diagram as shown. The relational or logical operation actions are on the transition from one junction to another. Relational statements specify conditions to conditionally allow a transition. In that case, the statement would be within square brackets.

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

Results

Code implementing the logical operator OR is in the `ex_logical_SF_step` function in `ex_logical_SF.c`:

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

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

Modeling Pattern for Relational and Logical Operators — MATLAB Function Block

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

Procedure

1 Follow the steps for "Set Up an Example Model With a MATLAB Function Block" on page 13-6 . This example model contains two Inport blocks and one Outport block.

2 Name the example model `ex_rel_operator_ML`.

3 In the MATLAB Function Block Editor enter the function, as follows:

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

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

Results

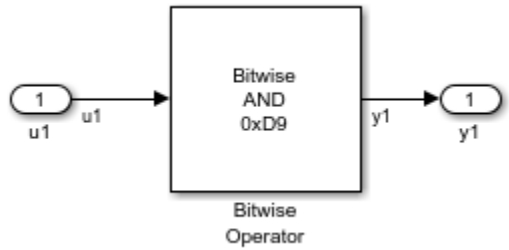
Code implementing the relational operator `'>'` is in the `ex_rel_operator_ML_step` function in `ex_rel_operator_ML.c`:

```
real_T u1;                          /* '<Root>/u1' */
real_T u2;                          /* '<Root>/u2' */
boolean_T y;                        /* '<Root>/MATLAB Function' */
```

```
void ex_rel_operator_ML_step(void)
{
  y = (u1 > u2);
 }
```

Bitwise Operations

Simulink Bitwise-Operator Block



ex_bit_logic_SL

Procedure

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

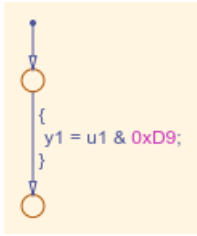
   Note: If another input uses Bitwise operations, clear the **Use bit mask** parameter and enter the number of input ports.
5. In the **Bit Mask** field, enter a decimal number. Use `bin2dec` or `hex2dec` to convert from binary or hexadecimal. In this example, enter `hex2dec('D9')`.
6. Name the blocks, as shown in, model `ex_bit_logic_SL`.
7. Connect the blocks and name the signals, as shown in, model `ex_bit_logic_SL`.
8. Press **Ctrl+B** to build the model and generate code.

Results

Code implementing the logical operator OR is in the `ex_bit_logic_SL_step` function in `ex_bit_logic_SL.c`:

```
uint8_T u1;
uint8_T y1;

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

Stateflow Chart



ex_bit_logic_SF/Bit_Logic Stateflow Chart

Procedure

1. Follow the steps for "Set Up an Example Model With a Stateflow Chart" on page 13-5. This example contains one Inport block and one Outport block.
2. Name the example model `ex_bit_logic_SF`.
3. From the **Stateflow Editor**, select **Tools > Explore** to open the Model Explorer.
4. In the Model Explorer, on the right pane, select **Enable C-bit operations**.
5. In the **Stateflow Editor**, create the Stateflow diagram, `ex_bit_logic_SF/Bit_Logic`.
6. Press **Ctrl+B** to build the model and generate code.

Results

Code implementing the logical operator OR is in the `ex_bit_logic_SF_step` function in `ex_bit_logic_SF.c`:

```
uint8_T u1;
uint8_T y1;

void bit_logic_SF_step(void)
```

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

MATLAB Function Block

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

Procedure

1. Follow the steps for "Set Up an Example Model With a MATLAB Function Block" on page 13-6. This example model contains two Inport blocks and one Outport block.
2. Name your model `ex_bit_logic_ML`.
3. In the MATLAB Function Block Editor enter the function, as follows:

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

   y1 = bitor(u1, u2);
   end
   ```
4. Press **Ctrl+B** to build the model and generate code.

Results

Code implementing the bitwise operator OR is in the `ex_bit_logic_ML_step` function in `ex_bit_logic_ML.c`:

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

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

Enumeration

To generate an enumerated data type, define an enumeration class in a MATLAB file. Then, use the enumeration class as the data type of signals, block parameters, and states in a model.

C Construct

```
typedef enum {
  Choice1 = 0,
  Choice2
} myEnumType;
```

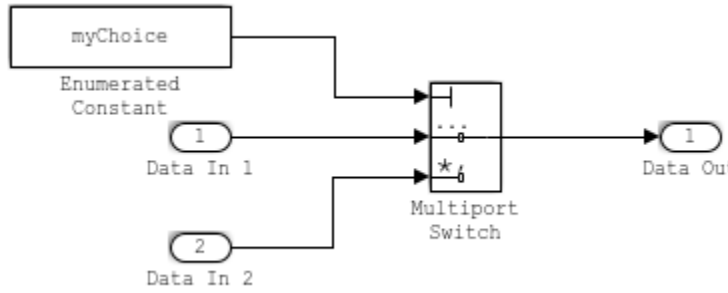
Procedure

In your current folder, create the MATLAB file `myEnumType.m`. The file defines an enumeration class `myEnumType`.

```
classdef myEnumType < Simulink.IntEnumType

    enumeration
        Choice1(0)
        Choice2(1)
    end %enumeration

    methods (Static)
        function retVal = getHeaderFile()
            retVal = 'myEnumHdr.h';
        end %function
        function retVal = getDataScope()
            retVal = 'Exported';
        end %function
    end %methods

end %classdef
```

Create the model `ex_pattern_enum` by using an Enumerated Constant block and a Multiport Switch block.

```
open_system('ex_pattern_enum')
```



In the base workspace, create a `Simulink.Parameter` object `myChoice`. Use the enumeration member `Choice1` to set the value of the parameter object.

```
myChoice = Simulink.Parameter(myEnumType.Choice1);
```

Set the storage class of the parameter object to `ExportedGlobal` so that the object appears in the generated code as a global variable.

```
myChoice.CoderInfo.StorageClass = 'ExportedGlobal';
```

In the Enumerated Constant block dialog box, set:

- **Output data type** to `Enum: myEnumType`.
- **Value** to `myChoice`.

In the Multiport Switch block dialog box, set:

- **Data port order** to `Specify indices`.
- **Data port indices** to `enumeration('myEnumType')`. This expression returns all of the enumeration members of `myEnumType`.

Generate code from the model.

```
rtwbuild('ex_pattern_enum');
```

```
### Starting build procedure for model: ex_pattern_enum
### Successful completion of build procedure for model: ex_pattern_enum
```

Results

View the generated header file `myEnumHdr.h`. The file defines the enumerated data type.

```
file = fullfile('ex_pattern_enum_ert_rtw','myEnumHdr.h');
rtwdemodbtype(file,'typedef enum {','} myEnumType;',1,1)
```

```
typedef enum {
  Choice1 = 0,                          /* Default value */
  Choice2
} myEnumType;
```

View the source file `ex_pattern_enum.c`. The file defines the variable `myChoice`. The algorithm in the `step` function uses `myChoice` to route one of the input signals to the output signal.

```
file = fullfile('ex_pattern_enum_ert_rtw','ex_pattern_enum.c');
rtwdemodbtype(file,'myEnumType myChoice = Choice1;','/* Variable: myChoice',1,1)
rtwdemodbtype(file,'/* Model step function */','/* Model initialize function */',1,0)
```

```
myEnumType myChoice = Choice1;          /* Variable: myChoice

/* Model step function */
void ex_pattern_enum_step(void)
{
  /* MultiPortSwitch: '<Root>/Multiport Switch' incorporates:
   *  Constant: '<S1>/Constant'
   */
  if (myChoice == Choice1) {
    /* Outport: '<Root>/Data Out' incorporates:
     *  Inport: '<Root>/Data In 1'
     */
    rtY.DataOut = rtU.DataIn1;
  } else {
    /* Outport: '<Root>/Data Out' incorporates:
     *  Inport: '<Root>/Data In 2'
     */
    rtY.DataOut = rtU.DataIn2;
  }

  /* End of MultiPortSwitch: '<Root>/Multiport Switch' */
}
```

See Also

`enumeration`

Related Examples

- "Use Enumerated Data in Simulink Models" (Simulink)
- "Use Enumerated Data in Generated Code" on page 19-22
- "Define Enumerated Data in a Chart" (Stateflow)
- "Define Enumerations for MATLAB Function Blocks" (Simulink)

If-Else

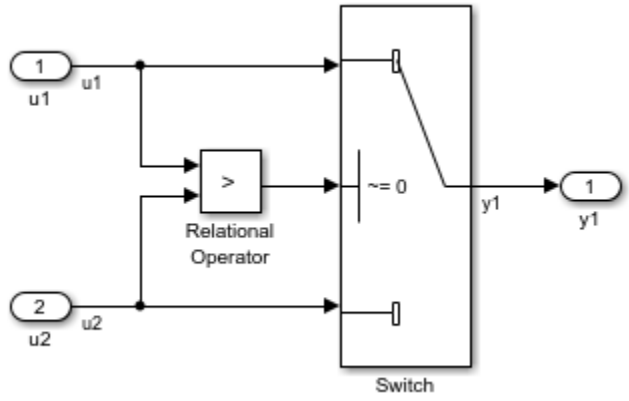
C Construct

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

Modeling Patterns

Modeling Pattern for If-Else: Switch block

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



Model ex_if_else_SL

Procedure

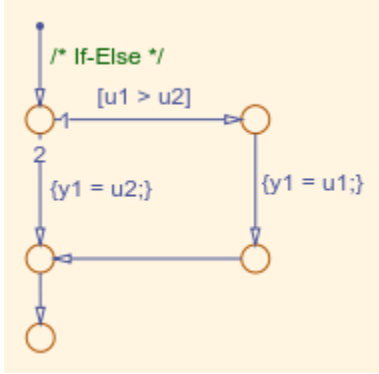
1. Drag the Switch block from the **Simulink>Signal Routing** library into your model.
2. Connect the data inputs and outputs to the block.
3. Drag a Relational Operator block from the Logic & Bit Operations library into your model.
4. Connect the signals that are used in the if-expression to the Relational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

Results

The generated code includes the following `ex_if_else_SL_step` function in the file `ex_if_else_SL.c`:

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

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

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

Modeling Pattern for If-Else: Stateflow Chart



ex_if_else_SF/Chart

Procedure

1. Follow the steps for "Set Up an Example Model With a Stateflow Chart" on page 13-5. This example model contains two Inport blocks and one Outport block.
2. Name your model `ex_if_else_SF`.
3. When configuring your Stateflow chart, select **Chart** > **Add Patterns** > **Decision** > **If-Else**. The Stateflow Pattern dialog opens. Fill in the fields as follows:

Description	`If-Else` (optional)
If condition	`u1 > u2`
If action	`y1 = u1`
Else action	`y1 = u2`

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

Results

The generated code includes the following `ex_if_else_SF_step` function in the file `If_Else_SF.c`:

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

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

```
/* Model step function */
void ex_If_Else_SF_step(void)
{
  /* Chart: '<Root>/Chart' incorporates:
   *  Inport: '<Root>/u1'
   *  Inport: '<Root>/u2'
   */
  /* Gateway: Chart */
  /* During: Chart */
  /* Entry Internal: Chart */
  /* Transition: '<S1>:15' */
  /*  If-Else */
  if (U.u1 > U.u2) {
    /* Outport: '<Root>/y1' */
    /* Transition: '<S1>:16' */
    /* Transition: '<S1>:18' */
    Y.y1 = U.u1;

    /* Transition: '<S1>:19' */
  } else {
    /* Outport: '<Root>/y1' */
    /* Transition: '<S1>:17' */
    Y.y1 = U.u2;
  }

  /* End of Chart: '<Root>/Chart' */
  /* Transition: '<S1>:20' */
}
```

Modeling Pattern for If-Else: MATLAB Function Block

Procedure

1. Follow the steps for "Set Up an Example Model With a MATLAB Function Block" on page 13-6. This example model contains two Inport blocks and one Outport block.
2. Name your model `ex_if_else_ML`.
3. In the MATLAB Function Block Editor enter the function, as follows:

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

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

Results

The generated code includes the following `ex_if_else_ML_step` function in the file `ex_if_else_ML.c`:

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

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

/* Model step function */
void ex_if_else_ML_step(void)
{
  /* MATLAB Function Block: '<Root>/MATLAB Function' incorporates:
   *  Inport: '<Root>/u1'
   *  Inport: '<Root>/u2'
   *  Outport: '<Root>/y1'
   */
  /* MATLAB Function 'MATLAB Function': '<S1>:1' */
  if (U.u1 > U.u2) {
    /* '<S1>:1:4' */
    /* '<S1>:1:5' */
    Y.y1 = U.u1;
  } else {
    /* '<S1>:1:6' */
    Y.y1 = U.u2;
  }
}
```

Switch

C Construct

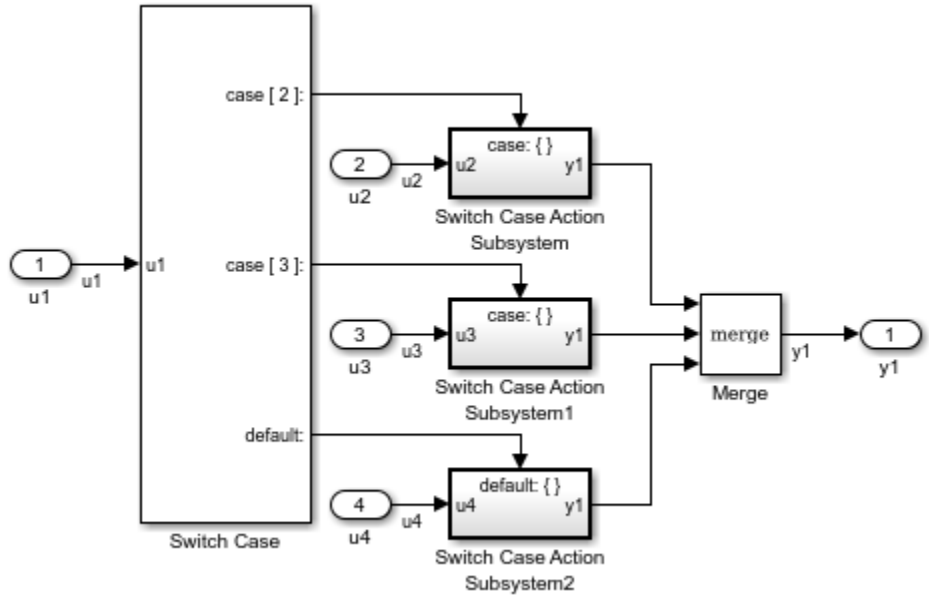
```
switch (u1)
{
 case 2:
   y1 = u2;
   break;
 case 3:
   u3;
   break;
 default:
   y1 = u4;
   break;
}
```

Modeling Patterns

- "Modeling Pattern for Switch: Switch Case block" on page 13-35
- "Modeling Pattern for Switch: MATLAB Function block" on page 13-38
- "Convert If-Elseif-Else to Switch statement" on page 13-39

Modeling Pattern for Switch: Switch Case block

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



Model ex_switch_SL

Procedure

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

6 Configure the Switch Case Action Subsystem (Simulink) subsystems.

7 Drag a Merge block from the **Simulink > Signal Routing** library to merge the outputs.

8 The Switch Case block takes an integer input, therefore, the input signal `u1` is type cast to an `int32`.

9 Enter `Ctrl+B` to build the model and generate code.

Results

The generated code includes the following `ex_switch_SL_step` function in the file `ex_switch_SL.c`:

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

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

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

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

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

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

```
    break;
  }
}
```

Modeling Pattern for Switch: MATLAB Function block

Procedure

1. Follow the steps for "Set Up an Example Model With a MATLAB Function Block" on page 13-6. This example model contains four Inport blocks and one Outport block.
2. Name your model `ex_switch_ML`.
3. In the MATLAB Function Block Editor enter the function, as follows:

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

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

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

Results

The generated code includes the following `ex_switch_ML_step` function in the file `ex_switch_ML.c`:

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

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

   /* Model step function */
   void ex_switch_ML_step(void)
   {
     /* MATLAB Function Block: '<Root>/MATLAB Function' incorporates:
      *  Inport: '<Root>/u1'
      *  Inport: '<Root>/u2'
      *  Inport: '<Root>/u3'
      *  Inport: '<Root>/u4'
      *  Outport: '<Root>/y1'
      */
     /* MATLAB Function 'MATLAB Function': '<S1>:1' */
     /* '<S1>:1:4' */
     switch (U.u1) {
      case 2:
       /* '<S1>:1:6' */
       Y.y1 = U.u2;
       break;
```

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

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

Convert If-Elseif-Else to Switch statement

If a MATLAB Function block or a Stateflow chart uses `if-elseif-else` decision logic, you can convert it to a `switch` statement by using a configuration parameter. In the Configuration Parameters dialog box, on the **Code Generation > Code Style** pane, select the "Convert if-elseif-else patterns to switch-case statements" parameter. For more information, see "Converting If-Elseif-Else Code to Switch-Case Statements" (Simulink). For more information on this conversion using a Stateflow chart, see "Enhance Readability of Code for Flow Charts" on page 36-127.

See Also

Switch Case

Related Examples

- "If-Else" on page 13-28
- "Enumeration" on page 13-24
- "Create Flow Charts with the Pattern Wizard" (Stateflow)
- "What Is a MATLAB Function Block?" (Simulink)

For Loop

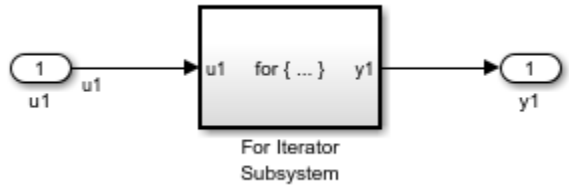
C Construct

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

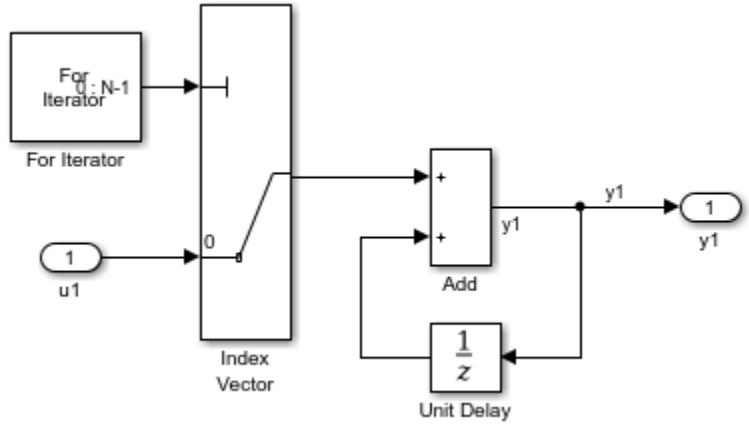
Modeling Patterns:

Modeling Pattern for For Loop: For-Iterator Subsystem block

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



Model ex_for_loop_SL



For Iterator Subsystem

Procedure

1. Drag a For Iterator Subsystem (Simulink) block from the **Simulink > Ports and Subsystems** library into your model.
2. Connect the data inputs and outputs to the For Iterator Subsystem block.
3. Open the Inport block.

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

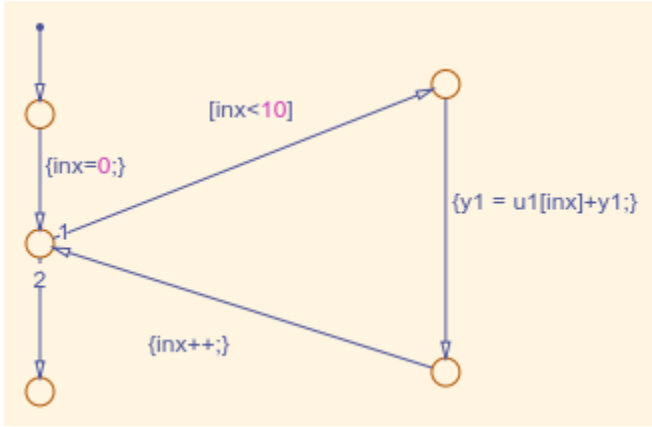
Results

The generated code includes the following `ex_for_loop_SL_step` function in the file `ex_for_loop_SL.c`:

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

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

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

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

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

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

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

Modeling Pattern for For Loop: Stateflow Chart



Model ex_for_loop_SF

Procedure

1. Follow the steps for "Set Up an Example Model With a Stateflow Chart" on page 13-5. This example model contains one Inport block and one Outport block.
2. Name the model `ex_for_loop_SF`.
3. Enter `Ctrl+R` to open the Model Explorer.
4. In the Model Explorer, select the output variable, `u1`, and in the right pane, select the **General** tab and set the **Initial Value** to `0`.
5. In the **Stateflow Editor**, select **Chart** > **Add Patterns** > **Loop** > **For**. The Stateflow Pattern dialog opens.
6. Fill in the fields in the Stateflow Pattern dialog box as follows:

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

The Stateflow diagram is shown.

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

Results

The generated code includes the following `ex_for_loop_SF_step` function in the file `ex_for_loop_SF.c`:

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

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

/* Model step function */
void ex_for_loop_SF_step(void)
{
  int32_T inx;

  /* Chart: '<Root>/Chart' */
  /* Gateway: Chart */
  /* During: Chart */
  /* Entry Internal: Chart */
  /* Transition: '<S1>:13' */
  /* Transition: '<S1>:14' */
  for (inx = 0; inx < 10; inx++) {
    /* Outport: '<Root>/y1' incorporates:
     *  Inport: '<Root>/u1'
     */
    /* Transition: '<S1>:11' */
    /* Transition: '<S1>:12' */
    Y.y1 += U.u1[inx];

    /* Transition: '<S1>:10' */
  }

  /* End of Chart: '<Root>/Chart' */
  /* Transition: '<S1>:9' */
}
```

Modeling Pattern for For Loop: MATLAB Function block

Procedure

1. Follow the directions for "Set Up an Example Model With a MATLAB Function Block" on page 13-6. This example model contains one Inport block and one Outport block.
2. Name your model `ex_for_loop_ML`.
3. In the MATLAB Function Block Editor enter the function, as follows:

```
function y1 = fcn(u1)

y1 = 0;

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

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

Results

The generated code includes the following `ex_for_loop_ML_step` function in the file `ex_for_loop_ML.c`:

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

/* Model step function */
void ex_for_loop_ML_step(void)
{
  int32_T inx;

  /* MATLAB Function Block: '<Root>/MATLAB Function' incorporates:
   *  Inport: '<Root>/u1'
   */
  /* MATLAB Function 'MATLAB Function': '<S1>:1' */
  /* '<S1>:1:3' */
  y1 = 0.0;
  for (inx = 0; inx < 10; inx++) {
    /* '<S1>:1:5' */
    /* '<S1>:1:6' */
    y1 = u1[inx] + y1;
  }
}
```

See Also

For Iterator Subsystem

Related Examples

- “Create Flow Charts with the Pattern Wizard” (Stateflow)

More About

- “What Is a MATLAB Function Block?” (Simulink)

While Loop

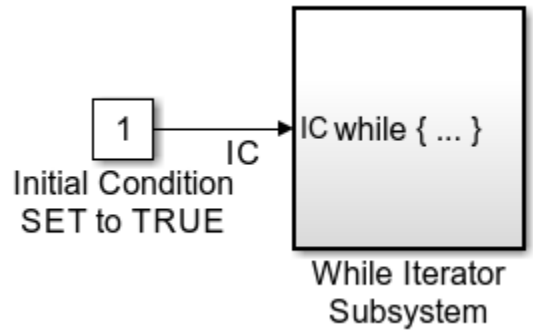
C Construct

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

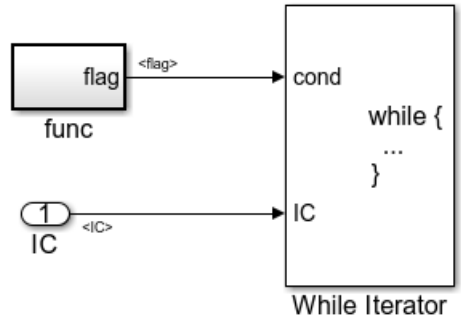
Modeling Patterns

Modeling Pattern for While Loop: While Iterator Subsystem block

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



Model ex_while_loop_SL

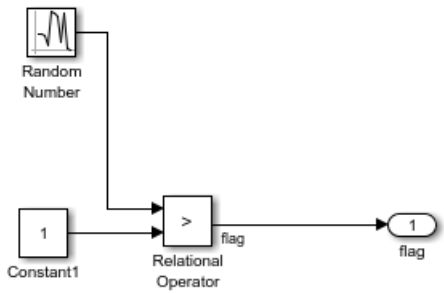


ex_while_loop_SL/While Iterator Subsystem

Procedure

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

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



func

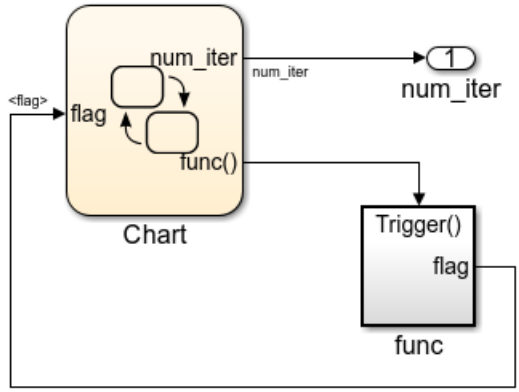
13. Double-click the While Iterator block to set the **Maximum number of iterations** to 100.
14. Connect blocks as shown in the model and subsystem diagrams.

Results

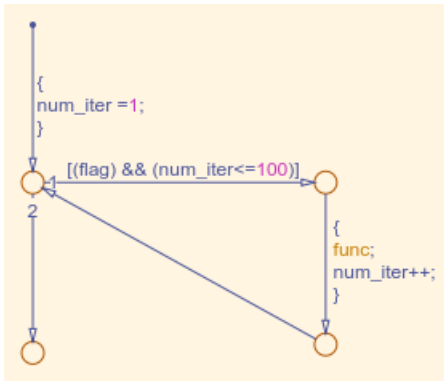
The generated code includes the following `ex_while_loop_SL_step` function in the file `ex_while_loop_SL.c`:

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

  /* Constant: '<Root>/Initial Condition SET to TRUE' */
  IC = P.InitialConditionSETtoTRUE_Value;

  /* Outputs for Iterator SubSystem: '<Root>/While Iterator Subsystem' incorporates:
   *  WhileIterator: '<S1>/While Iterator'
   */
  s1_iter = 1;

  /* InitializeConditions for Atomic SubSystem: '<S1>/func' */
  func_Init();

  /* End of InitializeConditions for SubSystem: '<S1>/func' */
  loopCond = IC;
  while (loopCond && (s1_iter <= 100)) {
    /* Outputs for Atomic SubSystem: '<S1>/func' */
    func();

    /* End of Outputs for SubSystem: '<S1>/func' */
    loopCond = flag;
    s1_iter++;
  }

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

Modeling Pattern for While Loop: Stateflow Chart



Model ex_while_loop_SF



ex_while_loop_SF/Chart Executes the desired while-loop

Procedure

1. Add a Stateflow Chart to your model from the **Stateflow > Chart** library.
2. Double-click the chart.
3. Add the input, `flag`, and output, `func`, to the chart and specify their data type.
4. Connect the data input and output to the Stateflow chart as shown in the model diagram.

5 In the Model Explorer, select the output variable, then, in the right pane, select the **General** tab and set the **Initial Value** to `0`.

6 Select **Chart** > **Add Patterns** > **Loop** > **While**. The Stateflow Pattern dialog opens.

7 Fill in the fields for the Stateflow Pattern dialog box as follows:

Description	`While Loop` (optional)
While condition	`(flag) && (num_iter<=100)`
Do action	`func; num_iter++;`

8 Place a Subsystem block in your model.

9 Right-click the subsystem and select **Block Parameters (Subsystem)**. The Block Parameters dialog box opens.

10 Select the **Treat as atomic unit** parameter to configure the subsystem to generate a function. This enables parameters on the **Code Generation** tab.

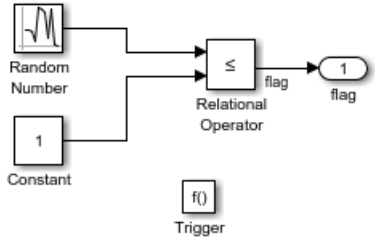
11 Select the **Code Generation** tab. From the **Function packaging** list, select the option, `Nonreusable function`.

12 From the **Function name options** list, select the option, `User specified`. The **Function name** parameter is displayed.

13 Specify the name as `func`.

14 Click **Apply** to apply the changes.

15 Double-click the `func` subsystem block. In this example, function `func` has an output `flag` set to `0` or `1` depending on the result of the algorithm in func( ). The Trigger block parameter **Trigger type** is `function-call`. Create the func() algorithm, as shown in the following diagram:



ex_while_loop_SF/func A function that updates the flag

16 Save and close the subsystem.

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

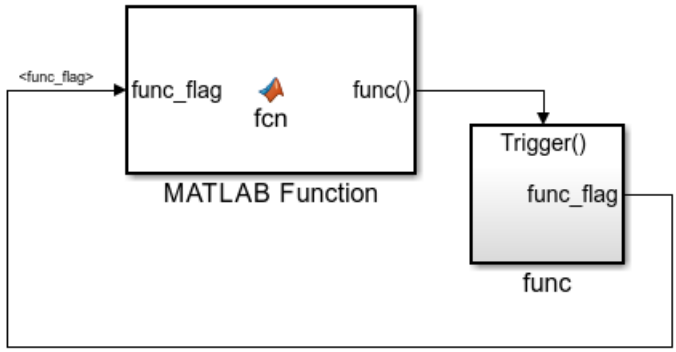
18 Save your model.

Results

The generated code includes the following `ex_while_loop_SF_step` function in the file `ex_while_loop_SF.c`:

```
/* Exported block signals */
int32_T num_iter;                      /* '<Root>/Chart' */
boolean_T flag;                        /* '<S2>/Relational Operator' */

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

/* Model step function */
void ex_while_loop_SF_step(void)
{
  /* Chart: '<Root>/Chart' */
  /* Gateway: Chart */
  /* During: Chart */
  /* Entry Internal: Chart */
  /* Transition: '<S1>:2' */
  num_iter = 1;
  while (flag && (num_iter <= 100)) {
    /* Outputs for Function Call SubSystem: '<Root>/func' */

    /* Transition: '<S1>:3' */
    /* Transition: '<S1>:4' */
    /* Event: '<S1>:12' */
    func();

    /* End of Outputs for SubSystem: '<Root>/func' */
    num_iter++;

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

  /* End of Chart: '<Root>/Chart' */
  /* Transition: '<S1>:1' */
}
```

Modeling Pattern for While Loop: MATLAB Function Block



Model ex_while_loop_ML

Procedure

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

```
function fcn(func_flag)

flag = true;
num_iter = 1;

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

4. Select **Edit Data** on the Editor tab. The Ports and Data Manager opens.
5. Select **Add > Function Call Output**. Change the name of the function call output to `func`.
6. Click **Save** and close the MATLAB Function Block Editor.

7 Place a Subsystem block in your model, right-click the subsystem and select **Block Parameters (Subsystem)**. The Block Parameters dialog box opens.

8 Select the **Treat as atomic unit** parameter to configure the subsystem to generate a function. This enables parameters on the **Code Generation** tab.

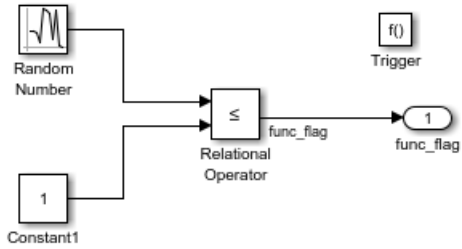
9 Select the **Code Generation** tab. From the **Function packaging** list, select the option, `Nonreusable function`.

10 From the **Function name options** list, select the option, `User specified`. The **Function name** parameter is displayed.

11 Specify the name as `func`.

12 Click **Apply**.

13 Double-click the `func()` subsystem block. In this example, function `func()` has an output `flag` set to `0` or `1` depending on the result of the algorithm in func( ). The Trigger block parameter **Trigger type** is `function-call`. Create the func() algorithm, as shown in the following diagram:



Model ex_while_loop_ML_func

14 Save and close the subsystem.

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

16 Save your model.

Results

The generated code includes the following `while_loop_ML_step` function in the file `while_loop_ML.c`. In some cases an equivalent `for` loop might be generated instead of a `while` loop.

```
/* Model step function */
void ex_while_loop_ML_step(void)
```

```
{
  boolean_T func_flag_0;
  boolean_T flag;
  int32_T num_iter;

  /* MATLAB Function: '<Root>/MATLAB Function' */
  func_flag_0 = func_flag;

  /* MATLAB Function 'MATLAB Function': '<S1>:1' */
  /* '<S1>:1:3' */
  flag = true;

  /* '<S1>:1:4' */
  num_iter = 1;
  while (flag && (num_iter <= 100)) {
    /* Outputs for Function Call SubSystem: '<Root>/func' */

    /* '<S1>:1:6' */
    /* '<S1>:1:7' */
    func();

    /* End of Outputs for SubSystem: '<Root>/func' */

    /* '<S1>:1:8' */
    flag = func_flag_0;

    /* '<S1>:1:9' */
    num_iter++;
  }

  /* End of MATLAB Function: '<Root>/MATLAB Function' */
}
```

See Also

While Iterator Subsystem

Related Examples

- “Create Flow Charts with the Pattern Wizard” (Stateflow)

More About

- “What Is a MATLAB Function Block?” (Simulink)

Do While Loop

C Construct

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

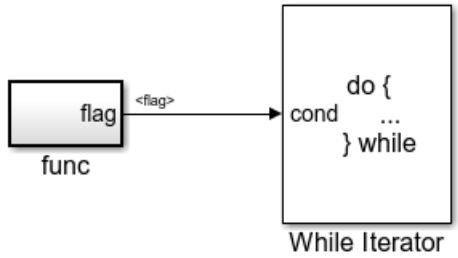
Modeling Patterns

- "Modeling Pattern for Do While Loop: While Iterator Subsystem block" on page 13-59
- "Modeling Pattern for Do While Loop: Stateflow Chart" on page 13-62

Modeling Pattern for Do While Loop: While Iterator Subsystem block

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

ex_do_while_loop_SL



ex_do_while_loop_SL/While Iterator Subsystem

Procedure

1. Drag a While Iterator Subsystem block from the **Simulink > Ports and Subsystems** library into the model.
2. Double-click the While Iterator Subsystem block to open the subsystem.
3. Place a Subsystem block next to the While Iterator block.
4. Right-click the subsystem and select **Block Parameters (Subsystem)**. The Block Parameters dialog box opens.
5. Select the **Treat as atomic unit** parameter to configure the subsystem to generate a function. This enables parameters on the **Code Generation** tab.

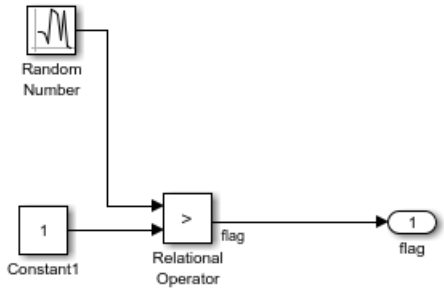
6 Select the **Code Generation** tab. From the **Function packaging** list, select the option, `Nonreusable function`.

7 From the **Function name options** list, select the option, `User specified`. The **Function name** parameter is displayed.

8 Specify the name as `func`.

9 Click **Apply**.

10 Double-click the `func` subsystem block. In this example, function `func` has an output `flag` set to `0` or `1` depending on the result of the algorithm in func. Create the `func` algorithm as shown in the following diagram:



ex_do_while_loop_SL/While Iterator Subsystem/func

11 Double-click the While Iterator block. This opens the Block Parameters dialog.

12 Set the **Maximum number of iterations** to 100.

13 Specify the **While loop type** as `do-while`.

14 Connect blocks as shown in the model and subsystem diagrams.

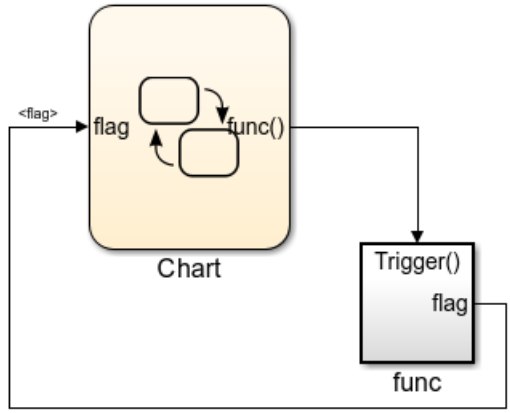
15 Enter `Ctrl+B` to generate code.

Results

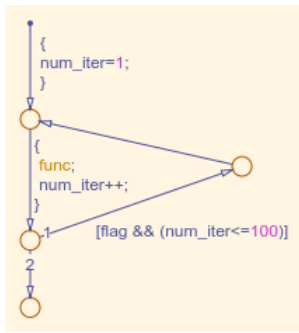
```
void func(void)
{
   flag = (DWork.NextOutput > (real_T)P.Constant1_Value);
   DWork.NextOutput =
      rt_NormalRand(&DWork.RandSeed) * P.RandomNumber_StdDev +
      P.RandomNumber_Mean;
}

void ex_do_while_loop_SL_step(void)
{
   int32_T s1_iter;
```

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

}
```

Modeling Pattern for Do While Loop: Stateflow Chart



ex_do_while_loop_SF



ex_do_while_loop_SF/Chart

1. Add a Stateflow Chart to your model from the **Stateflow > Chart** library.
2. Double-click the chart to open it.
3. Add the inputs and outputs to the chart and specify their data type.
4. Connect the data input and output to the Stateflow chart.
5. In the Model Explorer, select the output variable, then, in the right pane, select the **General** tab and set the **Initial Value** to 0.

6 Select **Chart > Add Patterns > Loop > While**. The Stateflow Pattern dialog opens.

7 Fill in the fields for the Stateflow Pattern dialog box as follows:

Description	`While Loop` (optional)
While condition	`(flag) && (num_iter<=100)`
Do action	`func; num_iter++;`

8 Place a Subsystem block in your model.

9 Right-click the subsystem and select **Block Parameters (Subsystem)**. The Block Parameters dialog box opens.

10 Select the **Treat as atomic unit** parameter to configure the subsystem to generate a function. This enables parameters on the **Code Generation** tab.

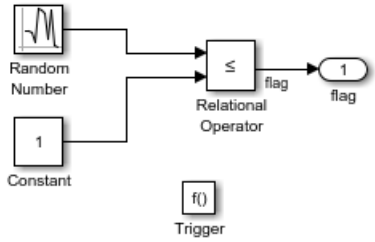
11 Select the **Code Generation** tab. From the **Function packaging** list, select the option, `Nonreusable function`.

12 From the **Function name options** list, select the option, `User specified`. The **Function name** parameter is displayed.

13 Specify the name as `func`.

14 Click **Apply** to apply the changes.

15 Double-click the `func` subsystem block. In this example, function `func` has an output `flag` set to `0` or `1` depending on the result of the algorithm in func. The Trigger block parameter **Trigger type** is `function-call`. Create the func algorithm, as shown in the following diagram:



ex_do_while_loop_SF/func Updates the flag

16 Save and close the subsystem.

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

18 Save your model.

Results

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

}
```

See Also

While Iterator Subsystem

Related Examples

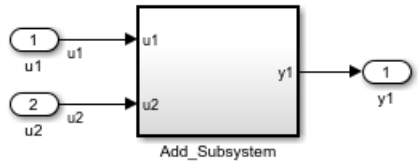
- “Create Flow Charts with the Pattern Wizard” (Stateflow)

Function Call

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

C Construct

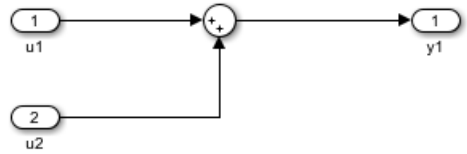
```
void add_function(void)
{
   y1 = u1 + u2;
}
```



ex_function_call

Procedure

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



ex_function_call/Add_Subsystem

3. Right-click the subsystem and select **Block Parameters (Subsystem)** to open the Subsystem Parameters dialog box.

4 Select the **Treat as atomic unit** parameter. This enables parameters on the **Code Generation** tab.

   Select the **Code Generation** tab. For the **Function packaging** parameter, from the drop-down list, select `Nonreusable function`.

5 For the **Function name options** parameter, from the drop-down list, select `User specified`.

6 In the **Function name** field, enter the subsystem name, `add_function`.

7 In the **File name options** field, select `Use function name`.

8 Click **Apply** and **OK**.

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

Results

In `ex_function_call.c`, the function is called from `ex_function_call_step`:

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

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

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

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

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

See Also

Function-Call Subsystem

Related Examples

- "Generate Reusable Function for Identical Subsystems Within a Model" (Simulink Coder)

More About

- "Conditional Subsystems" (Simulink)

Function Prototyping

C Construct

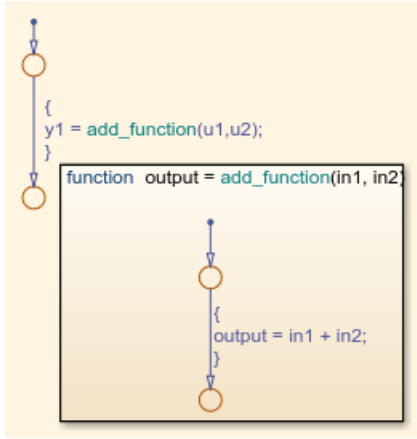
```
double add_function(double u1, double u2)
{
   return u1 + u2;
}
```

Modeling Patterns

Function Call Using Graphical Functions

Procedure

1. Follow the steps for "Set Up an Example Model With a Stateflow Chart" on page 13-5. This example model contains two Inport blocks and one Outport block.
2. Name the example model `ex_func_SF`.
3. In the **Stateflow Editor**, create a graphical function by clicking the ***fx*** button and placing a graphical function into the Stateflow chart.
4. Edit the graphical function signature to: `output = add_function(u1, u2)`.
5. Add the transition action, as shown in the following diagram.



ex_func_SF/Chart

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

6 Open the Model Explorer. From the Model Hierarchy tree, select **ex_func_SF > Chart > f()add_function**. On the right pane, specify the **Function Inline Option** as `Function`.

7 From the Model Hierarchy tree, click **Chart** and on the right pane select the **Export Chart Level Functions (Make Global)** parameter. This makes the function available globally to the entire model.

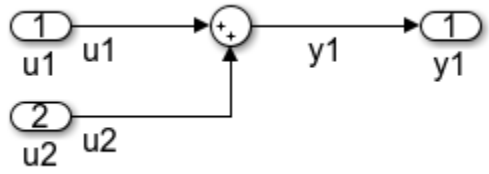
8 Press **Ctrl+B** to build the model and generate code.

Results

`ex_func_SF.c` contains the generated code:

```
real_T add_function(real_T in1, real_T in2)
{
   return in1 + in2;
}
.
.
.
void ex_func_SF_step(void)
{
   y1 = add_function(u1, u2);
}
```

Control Function Prototype of the *model*_step Function



ex_control_step_function

Procedure

1. Create the model, `ex_control_step_function`. See "Configure a Signal" on page 13-3 and "Configure Input and Output Ports" on page 13-4, for more information.
2. Press **Ctrl+E** to open the Configuration Parameters dialog box.
3. On the **Code Generation > Interface** pane, click **Configure Model Functions** to open the Model Interface dialog box.
4. Specify the **Function specification** parameter as `Model specific C prototypes`.
5. Click **Get Default Configuration** to update the **Configure model initialize and step functions** section and list the input and output arguments.
6. To configure the function output argument to pass a pointer, in the **Step function arguments** table, specify the **Category** for the Outport as a `Pointer`. In addition, you can specify the step function arguments order and type qualifiers.
7. To validate your changes, click **Validate**.
8. Press **Ctrl+B** to build the model and generate code.

Results

`ex_control_step_function.c` contains the generated code:

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

Related Examples

- "About Function Prototype Control" on page 26-2

- “Reuse Logic Patterns Using Graphical Functions” (Stateflow)

External C Functions

C Construct

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

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

Modeling Patterns

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

- "Use the Legacy Code Tool to Create S-functions" on page 13-71
- "Use a Stateflow Chart to Make Calls to C Functions" on page 13-73
- "Using a MATLAB Function Block to Make Calls to C Functions" on page 13-75

Use the Legacy Code Tool to Create S-functions

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

Procedure

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

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

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

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

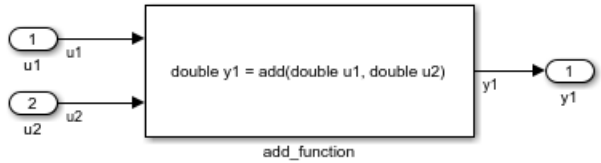
3 To build an S-function for use in both simulation and code generation, run the following script or execute each of these commands at the MATLAB command line:

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

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

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

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

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

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

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

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

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

The output of this script produces:

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

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



ex_function_call_lct

5 Name and save your model. In this example, the model is named `ex_function_call_lct`.

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

Results

The following code is generated in `ex_function_call_lct.c`:

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

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

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

Use a Stateflow Chart to Make Calls to C Functions

Procedure

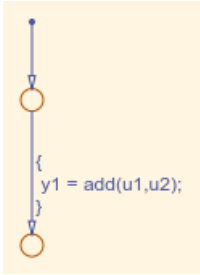
1 Create a C header file named `add.h` that contains the example function signature.

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

3 Follow the steps for "Set Up an Example Model With a Stateflow Chart" on page 13-5. This example model contains two Inport blocks and one Outport block.

4 Name the example model `ex_exfunction_call_SF`.

5 Double-click the Stateflow chart and edit the chart as shown. Place the call to the `add` function within a transition action.



ex_exfunction_call_SF/Chart

6 On the **Stateflow Editor**, select **Simulation** > **Model Configuration Parameters**.

7 On the Configuration Parameters dialog box, select **Simulation Target** > **Custom Code**. In the **Include custom C code in generated** section, on the left pane, select **Header file** and in the **Header file** field, enter the `#include` statement:

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

8 In the **Include list of additional** section, select **Source files** and in the **Source files** field, enter `add.c`.

9 On the Configuration Parameters dialog box, select **Code Generation** > **Custom Code**.

10 Select **Use the same custom code settings as Simulation Target** .

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

Results

`ex_exfunction_call_SF.c` contains the following code in the step function:

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

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

`ex_exfunction_call_SF.h` contains the include statement for `add.h`:

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

Using a MATLAB Function Block to Make Calls to C Functions

Procedure

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

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

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

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

end
```

6. Open the Configuration Parameters dialog box, and select **Simulation Target** > **Custom Code**.
7. In the **Include custom C code in generated** section, on the left pane, select **Header file** and in the **Header file** field, enter the statement, :

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

8. In the **Include list of additional** section, select **Source files** and in the **Source files** field, enter `add.c`.
9. Add two Inport blocks and one Outport block to the model and connect to the MATLAB Function block.
10. Configure the signals: `u1`, `u2`, and `y1`, as described in “Configure a Signal” on page 13-3.
11. Save the model as `ex_exfunction_call_ML`.
12. Press **Ctrl+B** to build the model and generate code.

Results

`ex_exfunction_call_ML.c` contains the following code:

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

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

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

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

Related Examples

- "Call C Functions in C Charts" (Stateflow)

More About

- "When to Generate Code from MATLAB Algorithms" (Simulink)
- "Legacy Code Tool and Code Generation" (Simulink Coder)

Macro Definitions (#define)

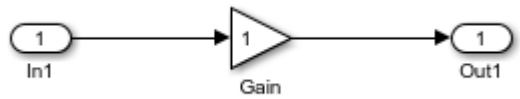
C Construct

```
#define myParam 9.8;
```

Export Generated Macro Definition

Procedure

1. Create the `ex_param_macro` model by using a Gain block.



2. In the Gain block dialog box, set **Gain** to `myParam`. Click **Apply**.
3. Click the button ⋮ next to the parameter value. Select **Create Variable**.
4. In the Create New Data dialog box, set **Value** to `Simulink.Parameter(9.8)`. Click **Create**.

   A `Simulink.Parameter` object, `myParam`, appears in the base workspace. The Gain block uses the object to set the value of the **Gain** parameter, in this case, 9.8.
5. In the Simulink.Parameter property dialog box, set **Storage class** to `Define`. Click **OK**.
6. Generate code from the model.

Results

The generated header file `ex_param_macro.h` defines `myParam` as a macro.

```
/* Definition for custom storage class: Define */
#define myParam                             9.8
```

Reuse Macro from Handwritten Code

Procedure

1. Follow the steps in “Export Generated Macro Definition” on page 13-77.

2 At the command prompt, change the custom storage class of `myParam` from `Define` to `ImportedDefine`.

```
myParam.CoderInfo.CustomStorageClass = 'ImportedDefine';
```

3 Configure the code generator to import the macro definition from a custom header file named `external_params.h`.

```
myParam.CoderInfo.CustomAttributes.HeaderFile = 'external_params.h';
```

4 In your current folder, create the C header file `external_params.h`, which contains the `#define` statement.

```
#ifndef _EXTERNAL_PARAMS
#define _EXTERNAL_PARAMS

#define myParam 9.8

#endif

/* EOF */
```

5 Generate code from the model.

Results

The generated header file `ex_param_macro.h` does not define the macro. Instead, the file includes (`#include`) the custom header file `external_params.h`.

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

The source file `ex_param_macro.c` contains a guard to check that a definition for `myParam` exists.

```
/*
* Check that imported macros with storage class "ImportedDefine" are defined
*/
#ifndef myParam
#error The variable for the parameter "myParam" is not defined
#endif
```

Related Examples

- “Exchange and Reuse Parameter Data Between Generated Code and Existing Code” on page 23-11

- “Control Data Representation by Applying Custom Storage Classes” on page 23-58

Conditional Inclusions (#if / #endif)

You can generate preprocessor conditional directives in your code by implementing variant blocks (Model Variants block or Variant Subsystem block) in your model. In the generated code, preprocessor conditional directives select a section of code to execute at compile time. To implement variants in your model, see “Create a Simple Variant Model” (Simulink). To generate code for variants, see “Generate Preprocessor Conditionals for Variant Systems” on page 14-33.

Typedef

C Construct

```
typedef float float_32;
```

Procedure

To create a data type alias in Simulink, use a `Simulink.AliasType`. The code generator creates a `typedef` statement.

The built-in Simulink data type `single` corresponds to the C data type `float`.

1. At the command prompt, create a `Simulink.AliasType` object named `float_32` that represents an alias of `single`.

   ```
   float_32 = Simulink.AliasType('single')
   ```

2. Create the `ex_typedef` model by using a Gain block.

3. In the model, select **View** > **Model Data**.
4. In the Model Data Editor, view the **Inports/Outports** tab.
5. From the **Change View** drop-down list, select `Design`.
6. In the model, select the Inport block.
7. In the Model Data Editor, for the Inport block, set **Data Type** to `float_32`.
8. From the **Change View** drop-down list, select `Code`.
9. For the Inport block, set **Signal Name** to `mySig`.
10. Set **Storage Class** to `ExportedGlobal`.

    With this setting, the Inport block appears in the generated code as a separate global variable.
11. Generate code from the model.

Results

The generated header file `ex_typedef.h` defines the data type alias `float_32`.

```
#ifndef DEFINED_TYPEDEF_FOR_float_32_
#define DEFINED_TYPEDEF_FOR_float_32_

typedef real32_T float_32;

#endif
```

By default, the code generator also creates the alias `real32_T`, which corresponds to the C data type `float`. You can see the `typedef` statement in the generated header file `rtwtypes.h`.

```
typedef float real32_T;
```

The generated source file `ex_typedef.c` uses `float_32` to define the global variable `mySig`.

```
/* Exported block signals */
float_32 mySig;                        /* '<Root>/In1' */
```

See Also

`Simulink.AliasType`

Related Examples

- “Create Data Type Alias in the Generated Code” on page 21-12
- “Data Type Replacement” on page 21-36
- “Structures of Signals” on page 13-87

Structures of Parameters

Create a structure in the generated code. The structure stores parameter data.

C Construct

```
typedef struct {
  double G1;
  double G2;
} myStructType;

myStructType myStruct = {
  2.0,
  -2.0
} ;
```

Procedure

At the command prompt, create a structure named `myStruct` with two fields.

```
myStruct.G1 = 2;
myStruct.G2 = -2;
```

Store the structure in a `Simulink.Parameter` object.

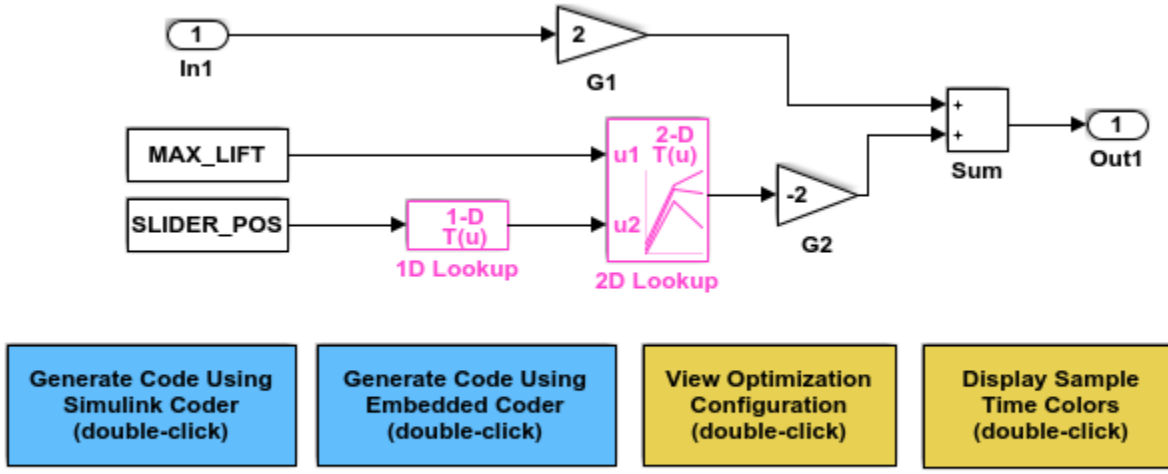
```
myStruct = Simulink.Parameter(myStruct);
```

Apply the storage class `ExportedGlobal` so that the structure appears in the generated code as a global variable.

```
myStruct.CoderInfo.StorageClass = 'ExportedGlobal';
```

Open the example model `rtwdemo_paraminline`.

```
rtwdemo_paraminline
```



In the G1 block dialog box, set **Gain** to `myStruct.G1`.

```
set_param('rtwdemo_paraminline/G1','Gain','myStruct.G1')
```

In the G2 block dialog box, set **Gain** to `myStruct.G2`.

```
set_param('rtwdemo_paraminline/G2','Gain','myStruct.G2')
```

Results

Generate code from the model.

```
rtwbuild('rtwdemo_paraminline')
```

```
### Starting build procedure for model: rtwdemo_paraminline
### Successful completion of build procedure for model: rtwdemo_paraminline
```

The generated header file `rtwdemo_paraminline_types.h` defines a structure type with a randomized name.

```
file = fullfile('rtwdemo_paraminline_grt_rtw',...
    'rtwdemo_paraminline_types.h');
```

```
rtwdemodbtype(file,'typedef struct {','} struct_6h72eH5WFuEIyQr5YrdGuB;',...
    1,1)
```

```
typedef struct {
  real_T G1;
  real_T G2;
} struct_6h72eH5WFuEIyQr5YrdGuB;
```

The source file `rtwdemo_paraminline.c` defines and initializes the structure variable `myStruct`.

```
file = fullfile('rtwdemo_paraminline_grt_rtw','rtwdemo_paraminline.c');
rtwdemodbtype(file,'struct_6h72eH5WFuEIyQr5YrdGuB myStruct',...
    '/* Variable: myStruct',1,1)
```

```
struct_6h72eH5WFuEIyQr5YrdGuB myStruct = {
  2.0,
  -2.0
} ;                                    /* Variable: myStruct
```

Specify Name of Structure Type

Optionally, specify a name to use for the structure type definition (`struct`).

Create a `Simulink.Bus` object that represents the structure type.

```
Simulink.Bus.createObject(myStruct.Value);
```

The default name of the object is `slBus1`. Change the name by copying the object into a new MATLAB variable.

```
myStructType = slBus1.copy;
```

Use the bus object as the data type of the parameter object.

```
myStruct.DataType = 'Bus: myStructType';
```

Generate code from the model.

```
rtwbuild('rtwdemo_paraminline')
```

```
### Starting build procedure for model: rtwdemo_paraminline
### Successful completion of build procedure for model: rtwdemo_paraminline
```

The code generates the definition of the structure type `myStructType` and uses this type to define the global variable `myStruct`.

```
rtwdemodbtype(file,'myStructType myStruct = {','/* Variable: myStruct',...
    1,1)
```

```
myStructType myStruct = {
  2.0,
  -2.0
} ;                                    /* Variable: myStruct
```

Related Examples

- "Organize Block Parameter Values into Structures in the Generated Code" on page 19-97

Structures of Signals

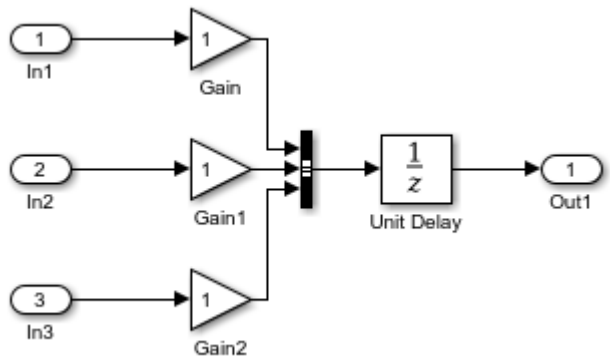
C Construct

```
typedef struct {
   double signal1;
   double signal2;
   double signal3;
} my_signals_type;
```

Procedure

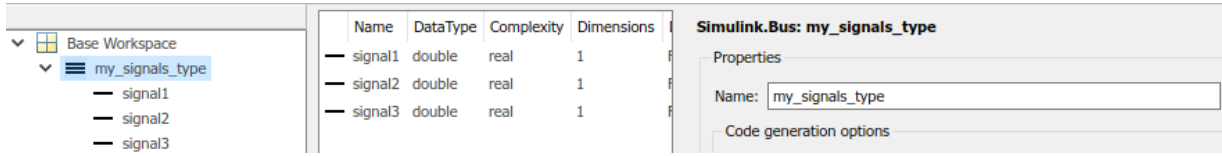
To represent a structure type in a model, create a `Simulink.Bus` object. Use the object as the data type of bus signals in your model.

1. Create the `ex_signal_struct` model with Gain blocks, a Bus Creator block, and a Unit Delay block. The Gain and Unit Delay blocks make the structure more identifiable in the generated code.



   To configure the Bus Creator block to accept three inputs, in the block dialog box, set **Number of inputs** to 3.

2. In the model, select **Edit** > **Bus Editor**.
3. Use the Bus Editor to create a `Simulink.Bus` object named `my_signals_type` that contains three signal elements: `signal1`, `signal2`, and `signal3`. To create bus objects with the Bus Editor, see "Create Bus Objects with the Bus Editor" (Simulink).



This bus object represents the structure type that you want the generated code to use.

4 In the Bus Creator block dialog box, set **Output data type** to `Bus: my_signals_type`.

5 Select **Output as nonvirtual bus**. Click **OK**.

A nonvirtual bus appears in the generated code as a structure.

6 In the model, select **View** > **Model Data**.

7 In the Model Data Editor, on the **Signals** tab, from the **Change View** drop-down list, select `Code`.

8 In the model, click the output signal of the Bus Creator block.

9 In the Model Data Editor, for the output of the Bus Creator block, set **Name** to `sig_struct_var`.

10 Set **Storage Class** to `ExportedGlobal`.

With this setting, the output of the Bus Creator block appears in the generated code as a separate global structure variable named `sig_struct_var`.

11 Generate code from the model.

Results

The generated header file `ex_signal_struct.h` defines the structure type `my_signals_type`.

```
typedef struct {
  real_T signal1;
  real_T signal2;
  real_T signal3;
} my_signals_type;
```

The source file `ex_signal_struct.c` allocates memory for the global variable `sig_struct_var`, which represents the output of the Bus Creator block.

```
/* Exported block signals */
my_signals_type sig_struct_var;        /* '<Root>/Bus Creator' */
```

In the same file, in the model `step` function, the algorithm uses `sig_struct_var` and the fields of `sig_struct_var`.

See Also

Simulink.Bus

Related Examples

- “Group Signals into Structures in the Generated Code Using Buses” on page 19-139
- “Combine Buses into an Array of Buses” (Simulink)

Nested Structures of Signals

C Construct

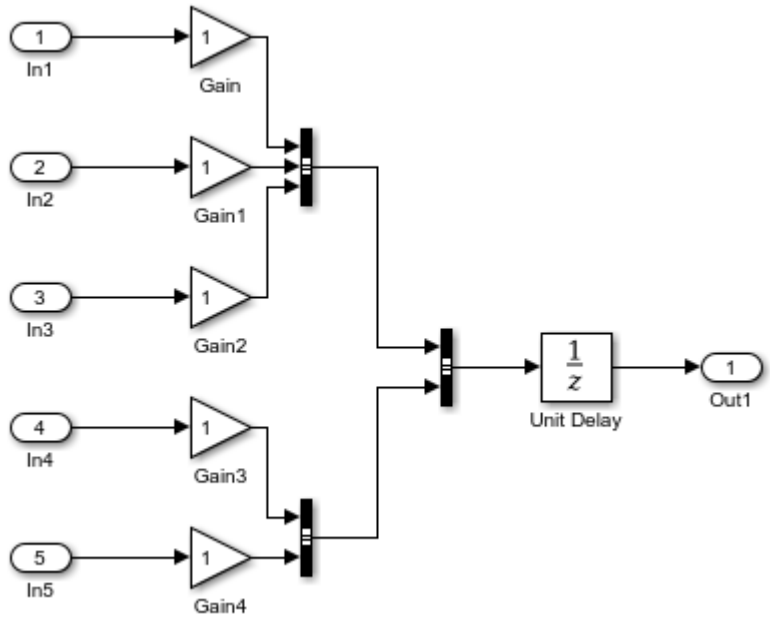
```
typedef struct {
  double signal1;
  double signal2;
  double signal3;
} B_struct_type;

typedef struct {
  double signal1;
  double signal2;
} C_struct_type;

typedef struct {
  B_struct_type subStruct_B;
  C_struct_type subStruct_C;
} A_struct_type;
```

Procedure

To represent a structure type in a model, create a `Simulink.Bus` object. Use the object as the data type of bus signals in your model.

To nest a structure inside another structure, use a bus object as the data type of a signal element in another bus object.

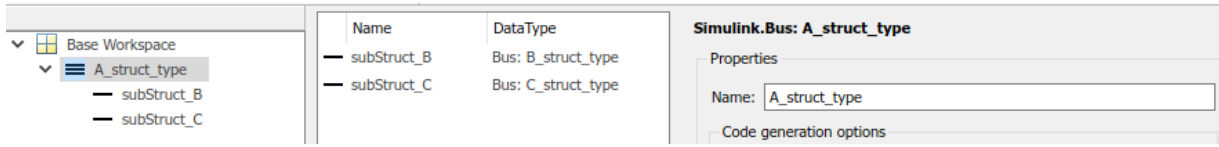
1. Create the `ex_signal_nested_struct` model with Gain blocks, Bus Creator blocks, and a Unit Delay block. The Gain and Unit Delay blocks make the structure more identifiable in the generated code.



To configure a Bus Creator block to accept three inputs, in the block dialog box, set **Number of inputs** to 3.

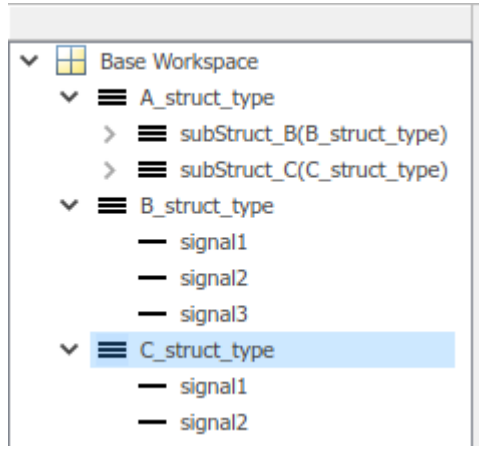
2. In the model, select **Edit** > **Bus Editor**.
3. Use the Bus Editor to create a `Simulink.Bus` object named `A_struct_type` that contains two signal elements: `subStruct_B` and `subStruct_C`. To create bus objects with the Bus Editor, see "Create Bus Objects with the Bus Editor" (Simulink).

   This bus object represents the top-level structure type that you want the generated code to use.
4. For the `subStruct_B` element, set **DataType** to `Bus: B_struct_type`. Use a similar type name for `subStruct_C`.



Each signal element in `A_struct_type` uses another bus object as a data type. Now these elements represent substructures.

5 Use the Bus Editor to create the `Simulink.Bus` objects `B_struct_type` (with three signal elements) and `C_struct_type` (with two signal elements).



6 In the dialog box of the Bus Creator block that collects the three Gain signals, set **Output data type** to `Bus: B_struct_type`. Click **Apply**.

7 Select **Output as nonvirtual bus**. Click **OK**.

8 In the dialog box of the other subordinate Bus Creator block, set **Output data type** to `Bus: C_struct_type` and select **Output as nonvirtual bus**. Click **OK**.

9 In the last Bus Creator block dialog box, set **Output data type** to `Bus: A_struct_type` and select **Output as nonvirtual bus**. Click **OK**.

10 In the model, select **View** > **Model Data**.

11 In the Model Data Editor, on the **Signals** tab, from the **Change View** drop-down list, select `Code`.

12 In the model, click the output signal of the `A_struct_type` Bus Creator block, which feeds the Unit Delay block.

13 In the Model Data Editor, for the output of the Bus Creator block, set **Name** to `sig_struct_var`.

14 Set **Storage Class** to `ExportedGlobal`.

   With this setting, the output of the Bus Creator block appears in the generated code as a separate global structure variable named `sig_struct_var`.

15 Generate code from the model.

Results

The generated header file `ex_signal_nested_struct.h` defines the structure types. Each structure type corresponds to a `Simulink.Bus` object.

```
typedef struct {
  real_T signal1;
  real_T signal2;
  real_T signal3;
} B_struct_type;

typedef struct {
  real_T signal1;
  real_T signal2;
} C_struct_type;

typedef struct {
  B_struct_type subStruct_B;
  C_struct_type subStruct_C;
} A_struct_type;
```

The generated source file `ex_signal_nested_struct.c` allocates memory for the global structure variable `sig_struct_var`. By default, the name of the `A_struct_type` Bus Creator block is Bus Creator2.

```
/* Exported block signals */
A_struct_type sig_struct_var;          /* '<Root>/Bus Creator2' */
```

In the same file, in the model `step` function, the algorithm uses `sig_struct_var` and the fields of `sig_struct_var`.

See Also

Simulink.Bus

Related Examples

- "Group Signals into Structures in the Generated Code Using Buses" on page 19-139
- "Combine Buses into an Array of Buses" (Simulink)

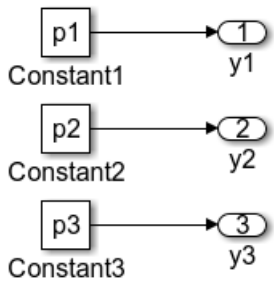
Bitfields

C Construct

```
typedef struct {
   unsigned int p1 : 1;
   unsigned int p2 : 1;
   unsigned int p3 : 1;
} my_struct_type
```

Procedure

1 Create the `ex_struct_bitfield_CSC` model with three Constant blocks and three Outport blocks. In each Constant block, set **Constant value** to `p1`, `p2`, or `p3`.



2 Create a `Simulink.Parameter` object in the base workspace for each Constant block, `p1`, `p2`, and `p3`. At the command prompt, enter:

```
p1 = Simulink.Parameter(false);
p2 = Simulink.Parameter(true);
p3 = Simulink.Parameter(false);
```

Each object stores a Boolean value (`true` or `false`) and uses the data type `boolean`.

3 Apply the custom storage class `BitField` to each parameter object.

```
p1.CoderInfo.StorageClass = 'Custom';
p1.CoderInfo.CustomStorageClass = 'BitField';

p2.CoderInfo.StorageClass = 'Custom';
p2.CoderInfo.CustomStorageClass = 'BitField';
```

```
p3.CoderInfo.StorageClass = 'Custom';
p3.CoderInfo.CustomStorageClass = 'BitField';
```

4 Configure each object to use the same structure type.

```
p1.CoderInfo.CustomAttributes.StructName = 'my_struct';

p2.CoderInfo.CustomAttributes.StructName = 'my_struct';

p3.CoderInfo.CustomAttributes.StructName = 'my_struct';
```

5 Generate code from the model.

Results

The generated header file `ex_struct_bitfield_CSC.h` defines the structure type `my_struct_type`.

```
/* Type definition for custom storage class: BitField */
typedef struct my_struct_tag {
  uint_T p1 : 1;
  uint_T p2 : 1;
  uint_T p3 : 1;
} my_struct_type;
```

The generated source file `ex_struct_bitfield_CSC.c` defines and initializes the structure variable `my_struct`.

```
/* Definition for custom storage class: BitField */
my_struct_type my_struct = {
  /* p1 */
  0,

  /* p2 */
  1,

  /* p3 */
  0
};
```

Related Examples

- “Control Data Representation by Applying Custom Storage Classes” on page 23-58

- “Pack Boolean data into bitfields” (Simulink)

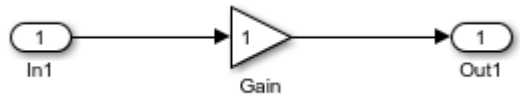
Arrays for Parameters

C Construct

```
float myParams[5]= {1.0F,2.0F,3.0F,4.0F,5.0F};
```

Procedure

1 Create the `ex_param_array` model by using a Gain block.



2 In the Gain block dialog box, set **Gain** to `myParams`. Click **Apply**.

3 Click the button ⋮ next to the parameter value. Select **Create Variable**.

4 In the Create New Data dialog box, set **Value** to `Simulink.Parameter([1 2 3 4 5])`. Click **Create**.

A `Simulink.Parameter` object, `myParams`, appears in the base workspace. The Gain block uses the object to set the value of the **Gain** parameter.

5 In the Simulink.Parameter property dialog box, set **Storage class** to `ExportedGlobal`.

With this setting, `myParams` appears in the generated code as a separate global variable.

6 Set **Data type** to `single`. Click **OK**.

7 Generate code from the model.

Results

The generated source file `ex_param_array.c` defines and initializes the global variable `myParams`.

```
/* Exported block parameters */
real32_T myParams[5] = { 1.0F, 2.0F, 3.0F, 4.0F, 5.0F } ;/* Variable: myParams
```

```
* Referenced by: '<Root>/Gain'
*/
```

Related Examples

- "Block Parameter Representation in the Generated Code" (Simulink Coder)
- "Code Generation of Matrices and Arrays" on page 33-76

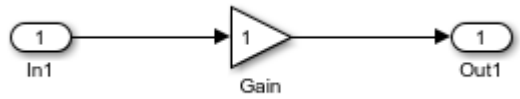
Arrays for Signals

C Construct

```
double myIn[5];
double myOut[5];
```

Procedure

1. Create the `ex_signal_array` model by using a Gain block.



2. In the model, select **View** > **Model Data**.
3. In the Model Data Editor, view the **Inports/Outports** tab.
4. From the **Change View** drop-down list, select `Design`.
5. In the model, select the Inport block.
6. In the Model Data Editor, for the Inport block, set **Signal Name** to `myIn`.
7. Set **Dimensions** to `[5 1]`.
8. For the Outport block, set **Signal Name** to `myOut`.
9. From the **Change View** drop-down list, select `Code`.
10. For the Inport block and the Outport block, set **Storage Class** to `ExportedGlobal`.

    With this setting, the blocks appear in the generated code as separate global variables.
11. Generate code from the model.

Results

The generated source file `ex_signal_array.c` defines the global variables `myIn` and `myOut` as arrays with 5 elements each.

```
/* Exported block signals */
```

```
real_T myIn[5];                         /* '<Root>/In1' */
real_T myOut[5];                        /* '<Root>/Out1' */
```

Related Examples

- "Determine Output Signal Dimensions" (Simulink)
- "Signal Dimensions" (Simulink)
- "Signal Representation in Generated Code" on page 19-112
- "Code Generation of Matrices and Arrays" on page 33-76

Pointers

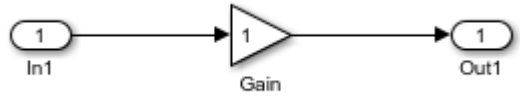
When your handwritten code allocates memory for signal, state, or parameter data, you can generate code that accesses that data through a pointer. Apply a storage class such as `ImportedExternPointer` to a data item in the model. Your handwritten code provides the pointer definition.

C Construct

```
extern double *myIn;
```

Procedure

1. Create the `ex_pointer` model by using a Gain block.



2. In the model, select **View** > **Model Data**.
3. In the Model Data Editor, view the **Inports/Outports** tab.
4. From the **Change View** drop-down list, select `Code`.
5. In the model, select the Inport block.
6. In the Model Data Editor, for the Inport block, set **Signal Name** to `myIn`.
7. Set **Storage Class** to `ImportedExternPointer`.
8. Generate code from the model.

Results

The generated header file `ex_pointer.h` declares the pointer.

```
/* Imported (extern) pointer block signals */
extern real_T *myIn;                   /* '<Root>/In1' */
```

In the generated source file `ex_pointer.c`, in the model `step` function, the algorithm dereferences the pointer, `myIn`.

```
/* Model step function */
void ex_pointer_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Inport: '<Root>/In1'
   */
  rtY.Out1 = *myIn;
}
```

Related Examples

- “Control Signals and States in Code by Applying Storage Classes” (Simulink Coder)
- “Block Parameter Representation in the Generated Code” on page 19-47

Variant Systems in Embedded Coder

Implement Dimension Variants for Array Sizes in Generated Code

Dimension Variants

Use symbolic dimensions to simulate various sets of dimension choices without regenerating code for every set. Set up your model with dimensions that you specify as symbols in blocks and data objects. These symbols propagate throughout the model during simulation, and then go into the generated code. Modeling constraints for symbols during simulation (for example, `C=A+B`) are output as preprocessor conditionals in either the `model.h` or the *`model`* `_types.h` file.

You can directly specify dimension information as a symbol or a numeric constant for these blocks and data objects:

- Inport
- Outport
- Signal Specification
- Data Store Memory
- Interpreted MATLAB Function
- `Simulink.Signal`
- `Simulink.Parameter`
- `Simulink.BusElement`
- `AUTOSAR.Parameter`

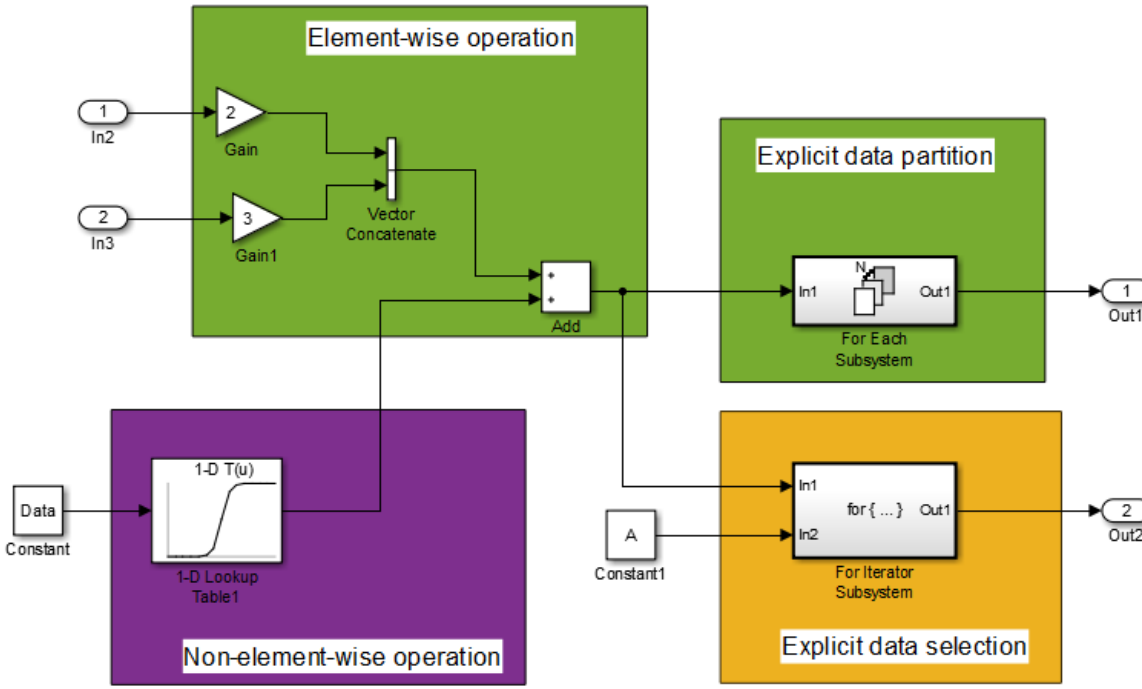
The Data Store Memory and Interpreted MATLAB Function blocks also support variable dimension signals. For these blocks, the symbolic dimensions control the maximum allowed size.

You use `Simulink.Parameter` objects to specify dimension information as symbols. For more information on signal dimensions, see “Signal Dimensions” (Simulink).

Note: The dimension variants feature is on by default. You can turn off this feature by clearing the “Allow symbolic dimension specification” (Simulink) parameter on the **All Parameters** tab of the Configuration Parameters dialog box.

Define Symbolic Dimensions

This example uses the model `rtwdemo_dimension_variants` to show how to implement symbolic dimensions. This model has four modeling patterns involving vectors and matrices.



1. Open the model matlab:rtwdemo_dimension_variants.
2. Open the Model Explorer. Select the base workspace pane.
3. In the base workspace, there are four `Simulink.Parameter` objects for specifying symbolic dimensions. These `Simulink.Parameter` objects have the names `A`, `B`, `C`, and `D`.
4. Select the `Simulink.Parameter` object `A`. Review the information in the Simulink.Parameter dialog box. `A` has a storage class of `CompilerFlag`.
5. Repeat Step 4 for each of the `Simulink.Parameter` objects `B`, `C`, and `D`.

6 For `Simulink.Parameter` objects with an `ImportedDefine` custom storage class, provide a header file on the MATLAB path. Insert the name of the header file in the **HeaderFile** field in the Simulink.Parameter dialog box.

To use a Simulink.Parameter object for dimension specification, it must have one of these storage classes:

- `Define` or `ImportedDefine` with header file specified
- `CompilerFlag`
- User-defined custom storage class that defines data as a macro in a specified header file

You can use MATLAB expressions to specify symbolic dimensions. For a list of supported MATLAB expressions, see the section Operators and Operands in Variant Condition Expressions in "Introduction to Variant Controls" (Simulink).

Specify Symbolic Dimensions for Blocks and Data Objects

1 Open the Source Block Parameters dialog box of Inport Block `In2`. In the **Signal Attributes** tab, the **Port Dimensions** field contains the `Simulink.Parameter` object `A`. For Inport blocks, you specify symbolic dimensions in the **Port Dimensions** field.

2 Open the Source Block Parameters dialog box of Inport block `In3`. In the **Signal Attributes** tab, the **Port Dimensions** field contains the `Simulink.Parameter` object `B`.

3 In the base workspace, select the `Simulink.Parameter` object `Data`. In the Simulink.Parameter dialog box for `Data`, the Dimension field has the character vector `'[1,C]'` , which is equivalent to `'[1,5]'` because `C` has a value of `5`. The **Value** field contains an array with `5` values, so the dimensions of `C` are consistent with the dimension of the `Data` object. The dimensions of the `Data` object must always be consistent with the value of the `Simulink.Parameter` object that is in the `Data` object **Dimensions** field. `Data` has a **Storage class** of `ImportedExtern`. A `Simulink.Parameter` object that uses a `Simulink.Parameter` for symbolic dimension specification must have a storage class of either `ImportedExtern` or `ImportedExternPointer`.

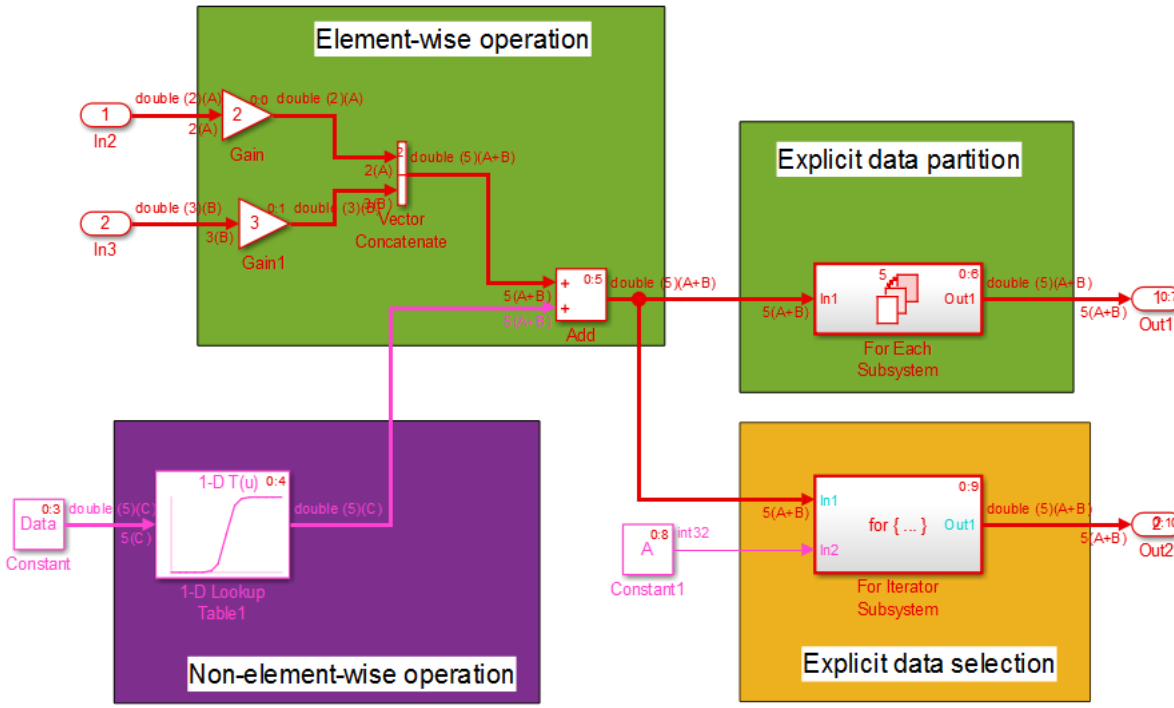
4 Open the Block Parameters dialog box of the `1-D Lookup Table1` block. The **Table data** field contains the `Simulink.Parameter`, `PT`. The **Breakpoints 1** field contains the `Simulink.Parameter`, `PB`.

5 In the base workspace, view the information in the Simulink.Parameter dialog boxes for `PB` and `PT`. These parameters contain the character vector `'[1,D]'` in their

Dimensions field and are arrays consisting of 15 values. The dimension of `D` are consistent with the dimension of the `PB` and `PT` parameters because `D` has a value of `15` .

6 Simulate the model. Simulink propagates the dimensions symbolically in the diagram. During propagation, Simulink establishes modeling constraints among symbols. Simulink then checks for consistency with these constraints based on current numerical assignments. One modeling constraint for `rtwdemo_dimension_variants` is that `C=A+B`. The **Diagnostic Viewer** produces a warning for any violations of constraints.

7 Change the dimension specification to a different configuration and simulate the model again.



Though not shown in this example, you can specify an n-D dimension expression with one or more of the dimensions being a symbol (for example, `'[A,B,C]'` or `'[1,A,3]'`).

Generate Code for a Model with Dimension Variants

Once you have verified dimension specifications through model simulation, generate code for `rtwdemo_dimension_variants`.

Create a temporary folder for the build and inspection process.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
```

Build the model.

```
model='rtwdemo_dimension_variants';
rtwbuild(model)
```

```
### Starting build procedure for model: rtwdemo_dimension_variants
### Successful completion of build procedure for model: rtwdemo_dimension_variants
```

View the generated code. In the `rtwdemo_dimension_variants.h` file, symbolic dimensions are in data declarations.

```
hfile = fullfile(cgDir,'rtwdemo_dimension_variants_ert_rtw',...
    'rtwdemo_dimension_variants.h');
rtwdemodbtype(hfile,'/* External inputs', '/* Real-time', 1, 0);
```

```
/* External inputs (root inport signals with auto storage) */
typedef struct {
  real_T In2[A];                       /* '<Root>/In2' */
  real_T In3[B];                       /* '<Root>/In3' */
} ExtU;

/* External outputs (root outports fed by signals with auto storage) */
typedef struct {
  real_T Out1[A + B];                  /* '<Root>/Out1' */
  real_T Out2[A + B];                  /* '<Root>/Out2' */
} ExtY;
```

The `rtwdemo_dimension_variants.h` file contains data definitions and preprocessor conditionals that define constraints established among the symbols during simulation. One of these constraints is that the value of a symbolic dimension must be greater than 1. This file also includes the user-provided header file for any `Simulink.Parameter` objects with an `ImportedDefine` custom storage class.

```
hfile = fullfile(cgDir,'rtwdemo_dimension_variants_ert_rtw',...
    'rtwdemo_dimension_variants.h');
```

```
rtwdemodbtype(hfile,'#ifndef A', '/* Macros for accessing', 1, 0);


#ifndef A
#error The variable for the parameter "A" is not defined
#endif

#ifndef B
#error The variable for the parameter "B" is not defined
#endif

#ifndef C
#error The variable for the parameter "C" is not defined
#endif

#ifndef D
#error The variable for the parameter "D" is not defined
#endif

/*
 * Constraints for division operations in dimension variants
 */
#if (1 == 0) || (((A+B) % 1) != 0)
# error "The preprocessor definition '1' must not be equal to zero and the division of
#endif

/*
 * Registered constraints for dimension variants
 */
/* Constraint 'C == (A+B)' registered by:
 * '<Root>/1-D Lookup Table1'
 */
#if C != (A+B)
# error "The preprocessor definition 'C' must be equal to '(A+B)'"
#endif

#if A <= 1
# error "The preprocessor definition 'A' must be greater than '1'"
#endif

#if B <= 1
# error "The preprocessor definition 'B' must be greater than '1'"
#endif

/* Constraint 'D > 1' registered by:
```

```
 * '<Root>/1-D Lookup Table1'
 */
#if D <= 1
# error "The preprocessor definition 'D' must be greater than '1'"
#endif

/* Constraint 'C > 1' registered by:
 * '<S2>/Assignment'
 */
#if C <= 1
# error "The preprocessor definition 'C' must be greater than '1'"
#endif
```

In the `rtwdemo_dimension_variants.c` file, symbolic dimensions participate in loop bound calculations, array size and index offset calculations, and a parameterized utility function (for example, Lookup Table block) calculation.

```
cfile = fullfile(cgDir,'rtwdemo_dimension_variants_ert_rtw',...
    'rtwdemo_dimension_variants.c');
rtwdemodbtype(cfile,'/* Model step', '/* Model initialize', 1, 0);
```

```
/* Model step function */
void rtwdemo_dimension_variants_step(void)
{
  /* local scratch DWork variables */
  int32_T ForEach_itr;
  int32_T iU;
  real_T rtb_VectorConcatenate[A + B];
  int32_T s2_iter;

  /* Gain: '<Root>/Gain' incorporates:
   *  Inport: '<Root>/In2'
   */
  for (iU = 0; iU <= (int32_T)(A - 1); iU++) {
    rtb_VectorConcatenate[iU] = 2.0 * rtU.In2[iU];
  }

  /* End of Gain: '<Root>/Gain' */

  /* Gain: '<Root>/Gain1' incorporates:
   *  Inport: '<Root>/In3'
   */
  for (iU = 0; iU <= (int32_T)(B - 1); iU++) {
    rtb_VectorConcatenate[(int32_T)(A + iU)] = 3.0 * rtU.In3[iU];
```

```
}

/* End of Gain: '<Root>/Gain1' */
for (iU = 0; iU <= (int32_T)(C - 1); iU++) {
  /* Sum: '<Root>/Add' incorporates:
   *  Constant: '<Root>/Constant'
   *  Lookup_n-D: '<Root>/1-D Lookup Table1'
   */
  rtb_VectorConcatenate[iU] += look1_binlx(Data[iU], PB, PT, (uint32_T)
    ((uint32_T)D - 1U));
}

/* Outputs for Iterator SubSystem: '<Root>/For Each Subsystem' incorporates:
 *  ForEach: '<S1>/For Each'
 */
for (ForEach_itr = 0; ForEach_itr < (int32_T)(A + B); ForEach_itr++) {
  /* ForEachSliceAssignment: '<S1>/ImpAsg_InsertedFor_Out1_at_inport_0' incorporates
   *  ForEachSliceSelector: '<S1>/ImpSel_InsertedFor_In1_at_outport_0'
   *  MATLAB Function: '<S1>/MATLAB Function'
   */
  /* MATLAB Function 'For Each Subsystem/MATLAB Function': '<S3>:1' */
  /* '<S3>:1:4' y = 2*u; */
  rtY.Out1[ForEach_itr] = 2.0 * rtb_VectorConcatenate[ForEach_itr];
}

/* End of Outputs for SubSystem: '<Root>/For Each Subsystem' */

/* Outputs for Iterator SubSystem: '<Root>/For Iterator Subsystem' incorporates:
 *  ForIterator: '<S2>/For Iterator'
 */
/* Constant: '<Root>/Constant1' */
for (s2_iter = 0; s2_iter < ((int32_T)A); s2_iter++) {
  /* Assignment: '<S2>/Assignment' incorporates:
   *  Constant: '<S2>/Constant'
   *  Outport: '<Root>/Out2'
   *  Product: '<S2>/Product'
   *  Selector: '<S2>/Selector'
   */
  if (s2_iter == 0) {
    for (iU = 0; iU <= (int32_T)((int32_T)(A + B) - 1); iU++) {
      rtY.Out2[iU] = rtb_VectorConcatenate[iU];
    }
  }
```

```
    rtY.Out2[s2_iter] = rtb_VectorConcatenate[s2_iter] * 2.0;

    /* End of Assignment: '<S2>/Assignment' */
  }

  /* End of Constant: '<Root>/Constant1' */
  /* End of Outputs for SubSystem: '<Root>/For Iterator Subsystem' */
}
```

Close the model and code generation report.

```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

Code Generation Optimization Considerations

When you create a model with symbolic dimensions, be aware of the following optimization considerations:

- The code generator reuses buffers only if dimension propagation establishes equivalence among buffers.
- Two loops with symbolic loop bound calculations are fused together only if they share equivalent symbolic expression.
- Optimizations do not eliminate a symbolic expression or condition check based on the current value of a symbolic dimension.

Backward Compatibility

If an existing model uses `Simulink.Parameter` objects to specify dimensions, it can be incompatible with dimension variants. Here are two common scenarios:

- Only a subset of blocks accepts symbolic dimension specifications. If a block is not compatible with symbolic dimensions, it causes an update diagram error.
- `Simulink.Parameter` objects that you use to define symbolic dimensions or have symbolic dimensions must have one of the storage classes described in this example. If these specifications are not met, the build procedure for the model fails during code generation.

You can address these backward compatibility issues by doing the following:

- Turn off dimension variants feature by clearing the **Allow symbolic dimension specification** parameter on the **All Parameters** tab in the Configuration Parameters dialog box.
- Update `Simulink.Parameter` objects that define symbolic dimensions or have symbolic dimension specifications.
- Update the model so that only supported blocks have symbolic dimensions or propagate symbolic dimensions.

Supported Blocks

For a list of supported blocks, see the Block Support Table. To access the information in this table, enter `showblockdatatypetable` at the MATLAB command prompt. Unsupported blocks (for example, MATLAB Function) can still work in a model containing symbolic dimensions as long as these blocks do not directly interact with symbolic dimensions.

In the following cases, supported blocks do not propagate symbolic dimensions.

- For `Unit Delay` blocks, you specify a `Simulink.Signal` object that has symbolic dimensions for the **Block Parameters** > **State Attributes** > **State name** parameter.
- For Assignment and Selector blocks, you set the **Block Parameters** > **Index Option** parameter to `Index vector (dialog)`. For Selector and Assignment blocks, if you specify a symbolic dimension for the Index parameter, the code generator does not honor the symbolic dimension in the generated code.
- For the Sum block, you specify `|+` for the **Block Parameters** > **List of signs** parameter, and you set the **Block Parameters** > **Sum over** parameter to `Specified dimension`.
- For the Product block, you specify a value of `1` for the **Block Parameters** > **Number of inputs** parameter, and you set the **Multiply over** parameter to **Specified dimensions**.
- For the ForEach block, you specify a symbolic dimension for the **Partition Width** parameter.

Note that the following modeling patterns are among those modeling patterns that can cause Simulink to error out:

- For Switch blocks, an input signal or the **Threshold** parameter has symbolic dimensions, and you select **Allow different data input sizes (Results in variable-size output signal)**.
- A Data Store Read block selects elements of a `Simulink.Bus` signal that has symbolic dimensions.
- For Lookup Table blocks, on the **Block Parameters** > **Algorithm** tab, you select the parameter **Use one input port for all input data**.

Limitations

The following products and software capabilities support dimension variants in that they act on the numeric value of a symbolic dimension. These features do not support the propagation of symbolic dimensions during model simulation and the preservation of symbolic dimensions in the generated code.

- Code Replacement for Lookup Tables
- Software-in-the-Loop (SIL) and Processor-in-the-Loop (PIL) simulations
- Accelerator and rapid accelerator simulation modes
- Scope and simulation observation (for example, logging, SDI, and so on)
- Model coverage
- Simulink Design Verifier
- Fixed-Point Designer
- Data Dictionary
- Simulink PLC Coder
- HDL Coder

The following do not support dimension variants:

- System Object
- Stateflow
- Physical modeling
- Discrete-event simulation
- Frame data
- MATLAB functions

The following limitations also apply to models that utilize symbolic dimensions.

- For simulation, the size of a symbolic dimension can equal 1. For code generation, the size of a symbolic dimension must be greater than `1`.
- If a symbolic dimension is a MATLAB expression that contains an arithmetic expression and either a relational or logical expression, you must add `+0` after the relational or logical part of the MATLAB expression. If you do not add `+0`, the model errors out during simulation because you cannot mix a `boolean` data type with integer or `double` data types. Adding `+0` converts the data type of the relational or logical part of the expression from a `boolean` to a `double`.

  For example, suppose in the Inport block parameters dialog box, the **Port dimensions** parameter has the expression `[(C==8)*D+E,3]`. The **Data type** parameter is set to `double`. Since `C==8` is a relational expression, you must change the expression to `[((C==8)+0)*D+E,3]` to prevent the model from producing an error during simulation.
- Simulink propagates symbolic dimensions for an entire structure or matrix, but not for a part of a structure or matrix. For example, the `Simulink.Parameter` `P` is a `2x3` matrix with symbolic dimensions `[Dim,Dim1]`.

  ```
  p=Simulink.Parameter(struct('A',[1 2 3;4 5 6]))
  p.DataType='Bus:bo'
  bo=Simulink.Bus
  bo.Elements(1).Name='A'
  bo.Elements(1).Dimensions='[Dim,Dim1]'
  Dim=Simulink.Parameter(2)
  Dim1=Simulink.Parameter(3)
  p.CoderInfo.StorageClass='Custom'
  p.CoderInfo.CustomStorageClass='Define'
  Dim.CoderInfo.StorageClass='Custom'
  Dim.CoderInfo.CustomStorageClass='Define'
  Dim1.Coderinfo.StorageClass='Custom'
  Dim1.CoderInfo.CustomStorageClass='Define'
  ```

  If you specify `p.A` for a dimensions parameter, Simulink propagates the symbolic dimensions `[Dim,Dim1]`. If you specify `p.A(1,:)`, Simulink propagates the numeric dimension `3` but not the symbolic dimension, `Dim1`.
- The MATLAB expression `A(:)` does not maintain symbolic dimension information. Use `A` instead.
- The MATLAB expression `P(2:A)` does not maintain symbolic dimension information. Use the Selector block instead.

- The MATLAB expression `P(2,:)` is not a tunable expression, so it does not maintain symbolic dimension information.
- Suppose that you set the value of a mask parameter, `myMaskParam`, by using a field of a structure or by using a subset of the structures in an array of structures. You store the structure or array of structures in a `Simulink.Parameter` object so that you can use a `Simulink.Bus` object to apply symbolic dimensions to the structure fields. Under the mask, you configure a block parameter to use one of the fields that have symbolic dimensions. The table shows some example cases.

Description	Value of mask parameter (`myMaskParam`)	Value of block parameter
`myStruct` is a structure with field `gains`, which uses symbolic dimensions.	`myStruct.gains`	`myMaskParam`
`myStruct` is a structure with field hierarchy `myStruct.subStruct.gai` The field `gains` uses symbolic dimensions.	`myStruct.subStruct`	`myMaskParam.gains`
`myStructs` is an array of structures. Each structure has a field `gains`, which uses symbolic dimensions.	`myStructs(2)`	`myMaskParam.gains`

In these cases, you cannot generate code from the model. As a workaround, choose one of these techniques:

- Use the entire structure (`myStruct`) or array of structures (`myStructs`) as the value of the mask parameter. Under the mask, configure the block parameter to dereference the target field from the mask parameter by using an expression such as `myMaskParam.subStruct.gains`.
- Use literal dimensions instead of symbolic dimensions for the target field (`gains`).

This limitation also applies when you use a field of a structure or a subset of the structures in an array of structures as the value of a model argument in a Model block.

Related Examples

- “Configure Dimension Variants for S-Function Blocks” on page 14-47

Code Generation for Variant Blocks

The code generator produces code from a Simulink model containing one or more Variant Subsystem, Variant Source, and Variant Sink blocks. To learn how to create a model containing variant blocks, see "Create a Simple Variant Model" (Simulink).

Code is generated for different variant choices, the active variant, and the default variant. To generate code for variants, set the following conditions in the Variant Subsystem, Variant Source, or Variant Sink block:

- Clear the option **Override variant conditions and use the following variant**.
- Select the option **Analyze all choices during update diagram and generate preprocessor conditionals**.

Code generated for Variant Subsystem blocks is surrounded by C preprocessor conditionals `#if`, `#else`, `#elif`, and `#endif`. Code generated for Variant Source and Variant Sink blocks is surrounded by C preprocessor conditionals `#if` and `#endif`. Therefore, the active variant is selected at compile time and the preprocessor conditionals determine which sections of the code to execute.

To construct model reference variants and generate preprocessor directives in the generated code, see the example "Use Model Variants to Generate Code That Uses C Preprocessor Conditionals".

To construct variant subsystems and generate preprocessor directives in the generated code, see the example "Use Subsystem Variants To Generate Code That Uses C Preprocessor Conditionals".

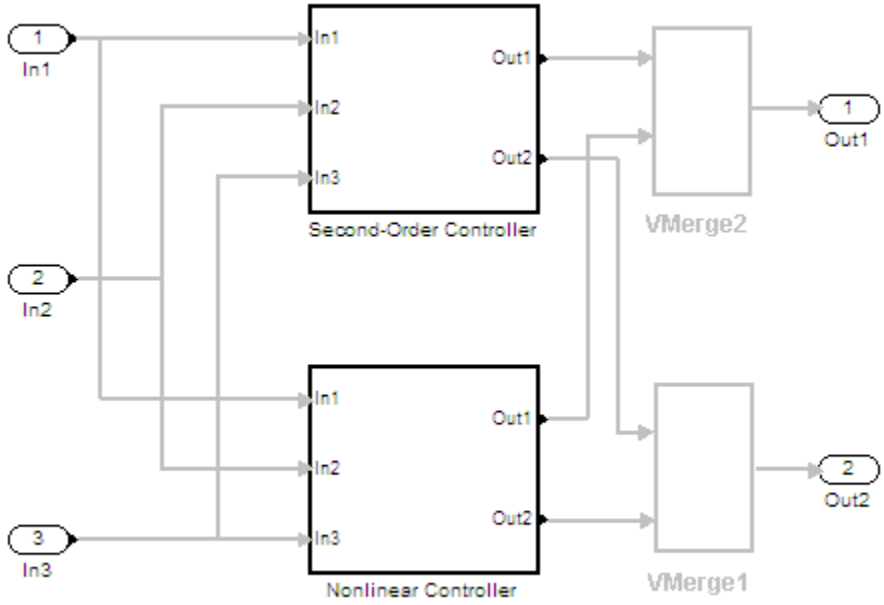
To construct models with variant sources and sinks and generate preprocessor directives in the generated code, see the example "Represent Variant Source and Sink Blocks in Generated Code" on page 14-37

Restrictions on Variant Subsystem Code Generation

To generate preprocessor conditionals, the types of blocks that you can place within the child subsystems of a Variant Subsystem block are limited. Connections are not allowed in the Variant Subsystem block diagram. However, during the code generation process, one `VariantMerge` block is placed at the input of each Outport block within the Variant Subsystem block diagram. All of the child subsystems connect to each of the `VariantMerge` blocks.

In the figure below, the code generation process makes the following connections and adds `VariantMerge` blocks to the sldemo_variant_subsystems model.

When compared to a generic Merge block the `VariantMerge` block can have only one parameter which is the number of Inputs. The `VariantMerge` block is used for code generation in variant subsystems internally, and is not available externally to be used in models. The number of inputs for `VariantMerge` is determined and wired as shown in the figure below.



The child subsystems of the Variant Subsystem block must be atomic subsystems. Select **Treat as atomic unit** parameter in the Subsystem block parameters dialog, to make the subsystems atomic. The `VariantMerge` blocks are inserted at the outport of the subsystems if more than one child subsystems are present. If the source block of a `VariantMerge` block input is nonvirtual, an error message will be displayed during code generation. You must make the source block contiguous, by inserting Signal Conversion blocks inside the variant choices. The signals that enter a Variant Subsystem block must have the same signal properties (for example, signal dimensions, port width, and storage class). The `VariantMerge` block does not support different signal properties because the

input ports and output ports share the same memory. You can use symbolic dimensions to generate code for a variant subsystem with child subsystems of different output signal dimensions.

Generated Code Components Not Compiled Conditionally

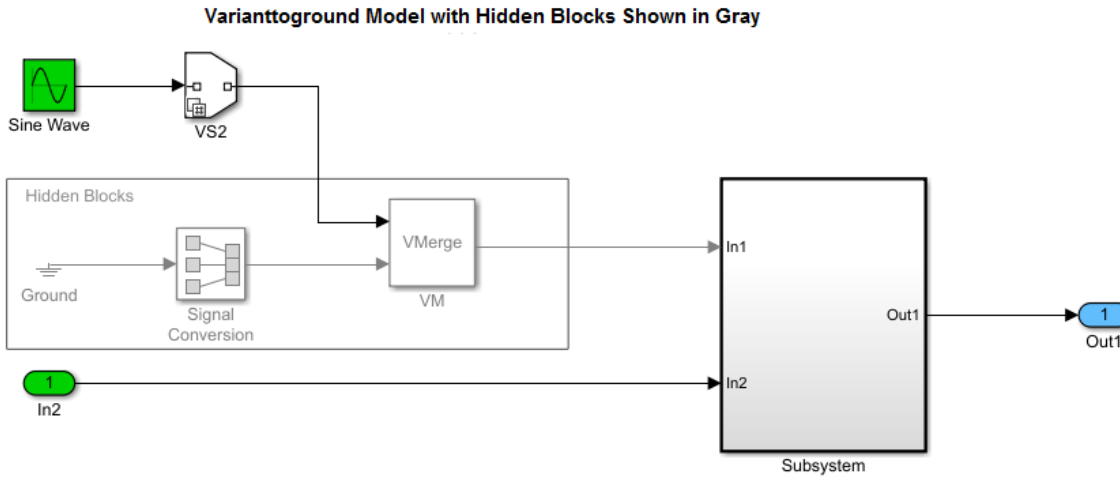
The following components are not conditionally compiled even if only code for variant subsystems or models that are conditionally compiled reference them.

- `rtModel` data structure fields
- `#include`'s of utility files
- Global non-constant parameter structure fields; when the configuration parameter **Optimization** > **Signals and Parameters** > **Parameter structure** is set to `NonHierarchical`
- Global constant parameter structure fields that are referenced by multiple subsystems activated by different variants
- Parameters that are configured to use an imported, exported, or custom code generation storage class, and are referenced by multiple subsystems that are activated by different variants
- Parameters that are configured to use an imported, exported, or custom code generation storage class, and are used by variant model blocks

Code Generation for Variant Blocks with One Variant Choice

For modeling patterns in which a Root Inport block connects to a Variant block with one variant choice, Simulink inserts a hidden block combination of a Ground block, Signal Conversion block, and a Variant Merge block. If the variant choice evaluates to false, this block combination produces an output of `0.0`.

For example, the model `Varianttoground` contains a Variant Source block with one variant choice. When the Variant Control `SYSCONST_A==6` evaluates to true, the input to `Subsystem` is a sine wave. When `SYSCONST_A==6` evaluates to false, the input to `Subsystem` is `0.0`.



The `varianttoground.c` file contains this code:

```
/* Sin: '<Root>/Sine Wave' */
#if SYSCONST_A == 6

  varianttoground_B.VM_Conditional_Signal_Subsystem_0_r64 = sin
    (varianttoground_M->Timing.t[0]);

#endif                                 /* SYSCONST_A == 6 */

  /* End of Sin: '<Root>/Sine Wave' */

  /* SignalConversion: '<Root>/VM_SignalConversion_Subsystem_0' */
#if SYSCONST_A != 6

  varianttoground_B.VM_Conditional_Signal_Subsystem_0_r64 = 0.0;

#endif                                 /* SYSCONST_A != 6 */

  /* End of SignalConversion: '<Root>/VM_SignalConversion_Subsystem_0' */
```

The comments in the generated code indicate the presence of the hidden signal conversion block. The code does not contain a comment for the Variant Merge block

because this block does not have associated generated code. The Variant Merge block is used internally and is not in the Simulink library.

Represent Subsystem and Model Variants in Generated Code

In this section...

Required products: Simulink, Embedded Coder, Simulink Coder

Using Simulink, you can create models that are based on a modular design platform that comprises a fixed common structure with a finite set of variable components. The variability helps you develop a single, fixed master design with variable components. For more information, see "What Are Variants and When to Use Them" (Simulink) (Simulink). When you implement variants in the generated code, you can:

- Reuse generated code from a set of application models that share functionality with minor variations.
- Share generated code with a third party that activates one of the variants in the code.
- Validate the supported variants for a model and then choose to activate one variant for a particular application, without regenerating and re-validating the code.
- Generate code for the default variant that is selected when an active variant does not exist.

Using Embedded Coder, you can generate code from Simulink models containing one or more variant choices. The generated code contains preprocessor conditionals that control the activation of each variant choice.

This example shows how to represent variant choices in a Simulink model and then prepare the model so that those variant choices are represented in generated code.

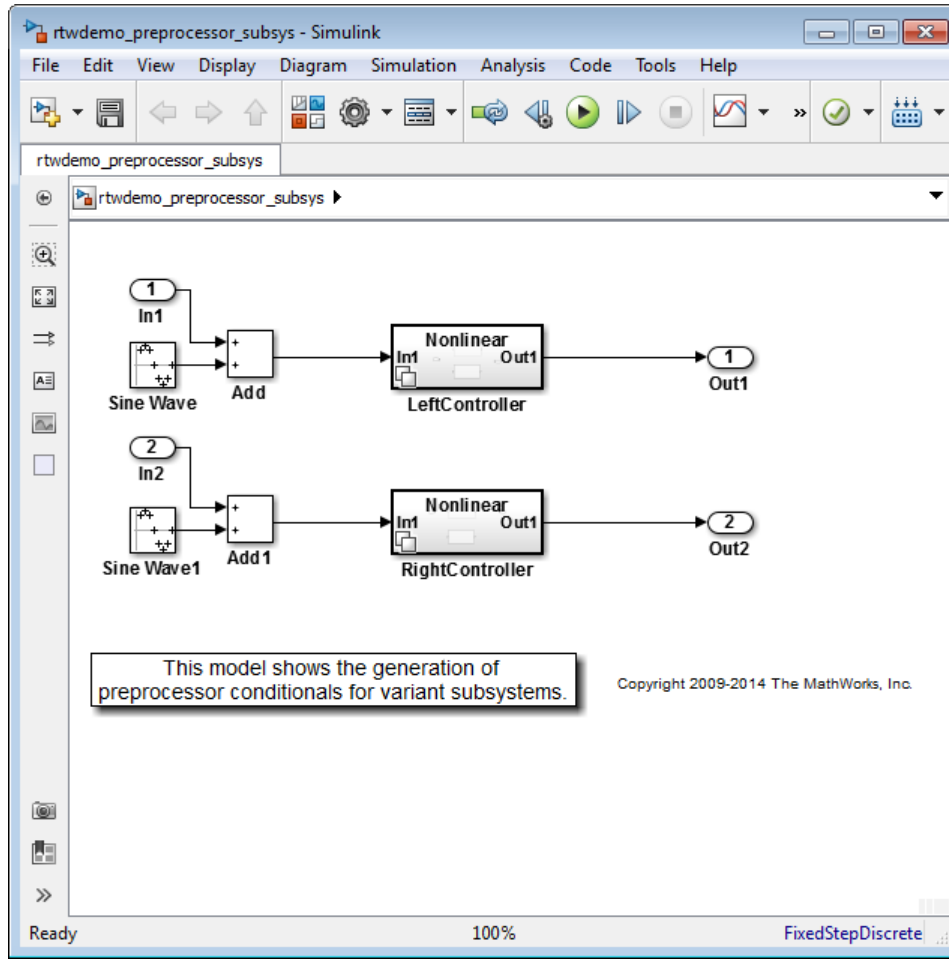
Step 1: Represent Variant Choices in Simulink

Variant choices are two or more configurations of a component in your model. This example uses the model `rtwdemo_preprocessor_subsys` to illustrate how to represent

variant choices inside Variant Subsystem blocks. For other ways to represent variant choices, see “Options for Representing Variants in Simulink” (Simulink) (Simulink).

1 Open the model `rtwdemo_preprocessor_subsys`.

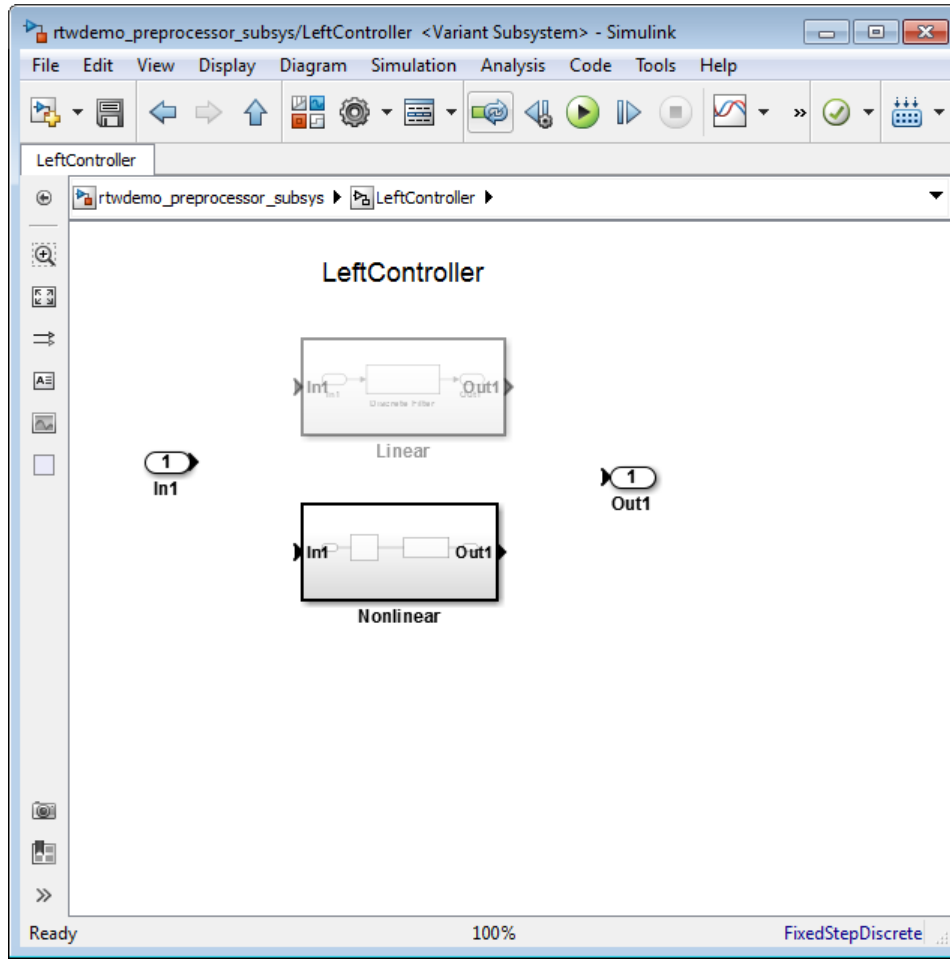
```
open_system('rtwdemo_preprocessor_subsys')
```



The model contains two Variant Subsystem blocks: **LeftController** and **RightController**.

Note: You can only add Inport, Outport, Subsystem, and Model blocks inside a Variant Subsystem block.

2 Open the **LeftController** block.



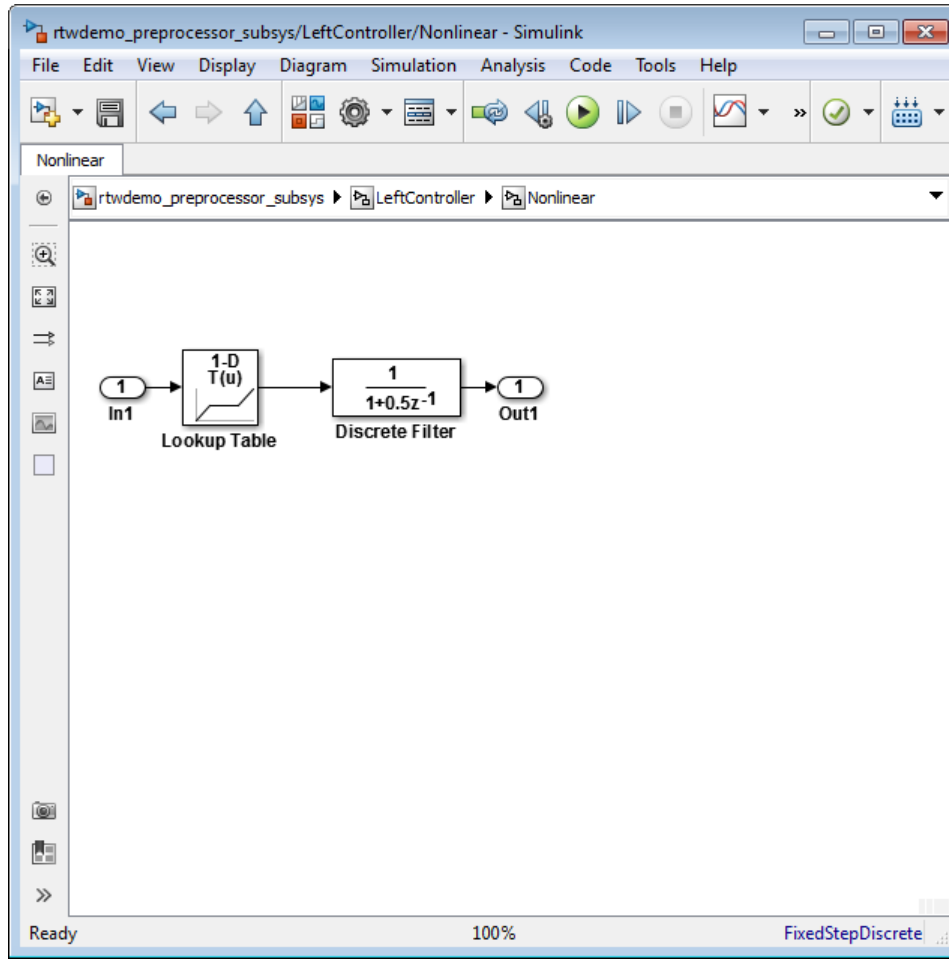
The **LeftController** block serves as the container for the variant choices. It contains two variant choices represented using Subsystem blocks **Nonlinear** and **Linear**.

The nonlinear controller subsystems implement hysteresis, whereas the linear controller subsystems act as simple low-pass filters.

The Subsystem blocks have the same number of inports and outports as the containing Variant Subsystem block.

Variant choices can have different numbers of inports and outports. See “Mapping Inports and Outports of Variant Choices” (Simulink) (Simulink).

3 Open the **Nonlinear** block.



The **Nonlinear** block represents one variant choice that Simulink activates when a condition is satisfied. The **Linear** block represents another variant choice.

Tip: When you are prototyping variant choices, you can create empty Subsystem blocks with no inputs or outputs inside a Variant Subsystem block. The empty subsystem recreates the situation in which that subsystem is inactive without the need for completely modeling the variant choice.

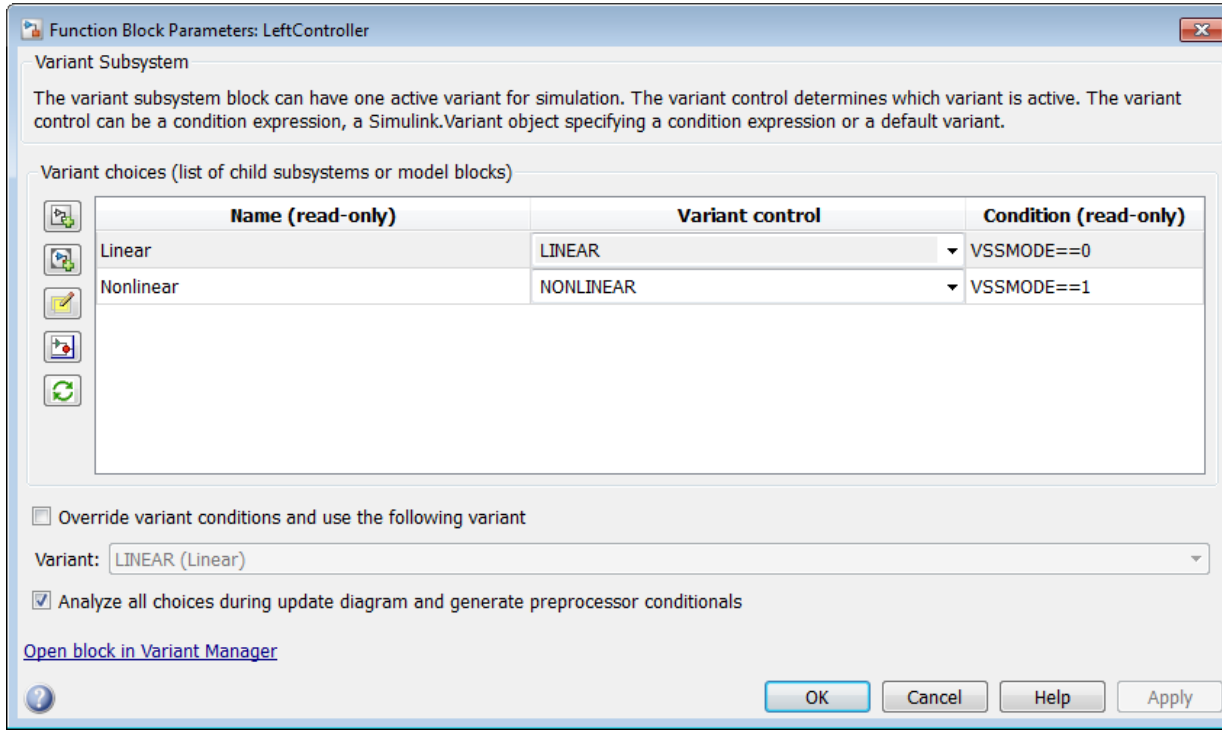
Step 2: Specify Conditions That Control Variant Choice Selection

You can switch between variant choices by constructing conditional expressions called variant controls for each variant choice represented in a Variant Subsystem block. Variant controls determine which variant choice is active, and changing the value of a variant control causes the active variant choice to switch.

A variant control is a Boolean expression that activates a specific variant choice when it evaluates to `true`.

For more information, see "Introduction to Variant Controls" (Simulink) (Simulink).

1 Right-click the **LeftController** block and select **Block Parameters (Subsystem)**.



The **Condition** column displays the Boolean expression that when `true` activates each variant choice. In this example, these conditions are specified using `Simulink.Variant` objects `LINEAR` and `NONLINEAR`.

2 Use these commands to specify a variant control using a `Simulink.Variant` object.

```
LINEAR = Simulink.Variant;
LINEAR.Condition = 'VSSMODE==0';
NONLINEAR = Simulink.Variant;
NONLINEAR.Condition = 'VSSMODE==1';
```

Here, `VSSMODE` is called a variant control variable that can be specified in one of the ways listed in "Approaches for Specifying Variant Controls" (Simulink) (Simulink).

3 Define the variant control variable `VSSMODE`.

You can define `VSSMODE` as a scalar variable. However, to generate code, specify variant control variables as `Simulink.Parameter` objects. In addition to enabling the specification of parameter value, `Simulink.Parameter` objects allow you to specify other attributes such as data type that are required for generating code.

```
VSSMODE = Simulink.Parameter;
VSSMODE.Value = 1;
VSSMODE.DataType = 'int32';
VSSMODE.CoderInfo.StorageClass = 'Custom';
VSSMODE.CoderInfo.CustomStorageClass = 'ImportedDefine';
VSSMODE.CoderInfo.CustomAttributes.HeaderFile = 'rtwdemo_importedmacros.h';
```

Variant control variables defined as `Simulink.Parameter` objects can have one of these storage classes.

- `Define` or `ImportedDefine` with header file specified
- `CompilerFlag`
- `SystemConstant (AUTOSAR)`
- Your own custom storage class that defines data as a macro

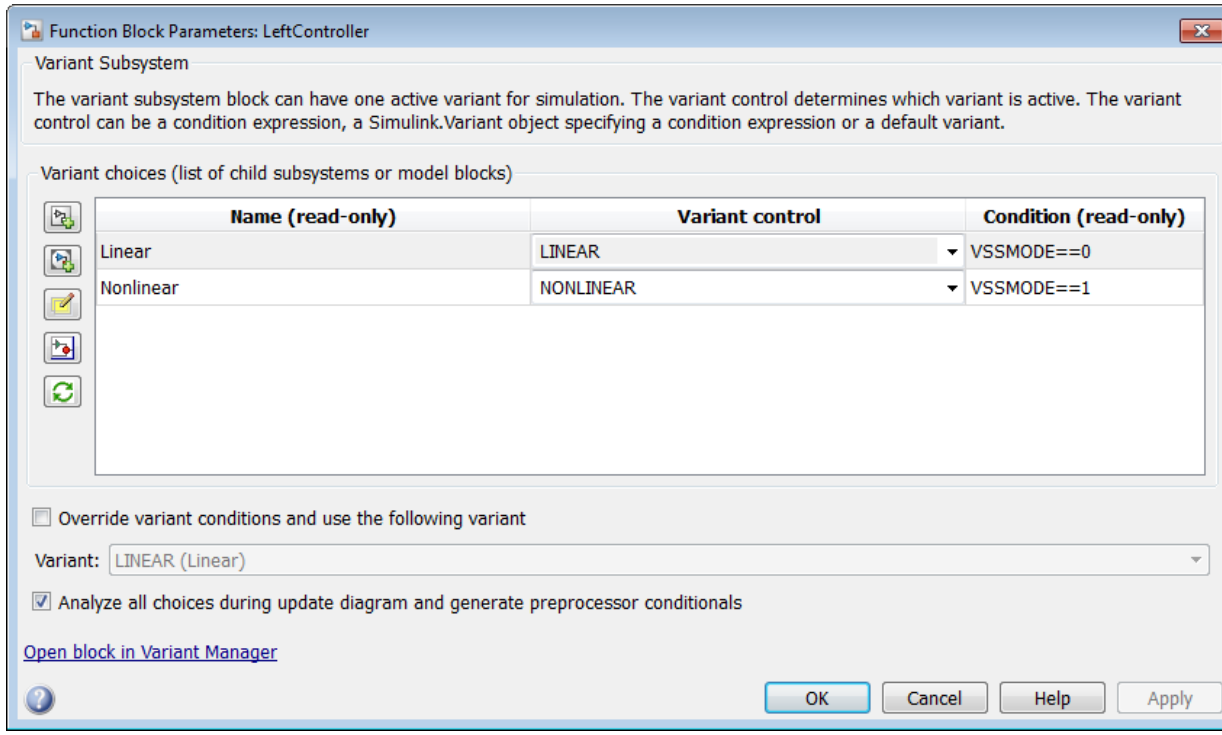
You can also convert a scalar variant control variable into a `Simulink.Parameter` object. See "Convert Variant Control Variables into Simulink.Parameter Objects" (Simulink) (Simulink).

Step 3: Configure Model for Generating Preprocessor Conditionals

Code generated for each variant choice is enclosed within C preprocessor conditionals `#if`, `#else`, `#elif`, and `#endif`. Therefore, the active variant is selected at compile time and the preprocessor conditionals determine which sections of the code to execute.

1. In the Simulink editor, select **Simulation** > **Model Configuration Parameters**.
2. Select the **Code Generation** pane, and set **System target file** to `ert.tlc`.
3. In the **Report** pane, select **Create code generation report**.
4. On the **All Parameters** tab of the Configuration Parameters dialog box, clear **Ignore custom storage classes** and click **Apply**.
5. In your model, right-click the **LeftController** block and select **Block Parameters (Subsystem)**.

6 Select the option **Analyze all choices during update diagram and generate preprocessor conditionals**.



When you select this option, Simulink analyzes all variant choices during an update diagram or simulation. This analysis provides early validation of the code generation readiness of all variant choices.

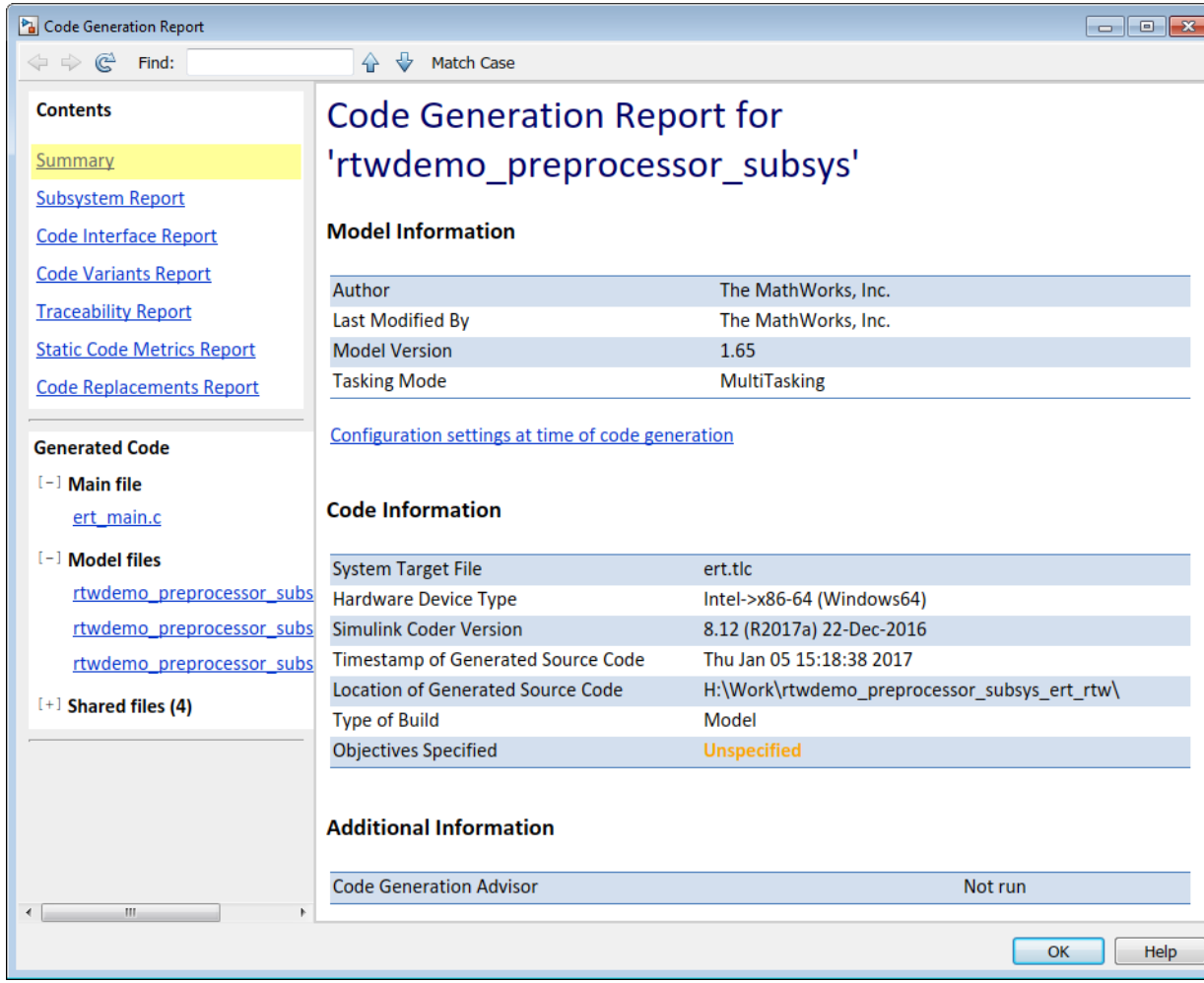
7 Clear the option **Override variant conditions and use following variant**.

8 Build the model.

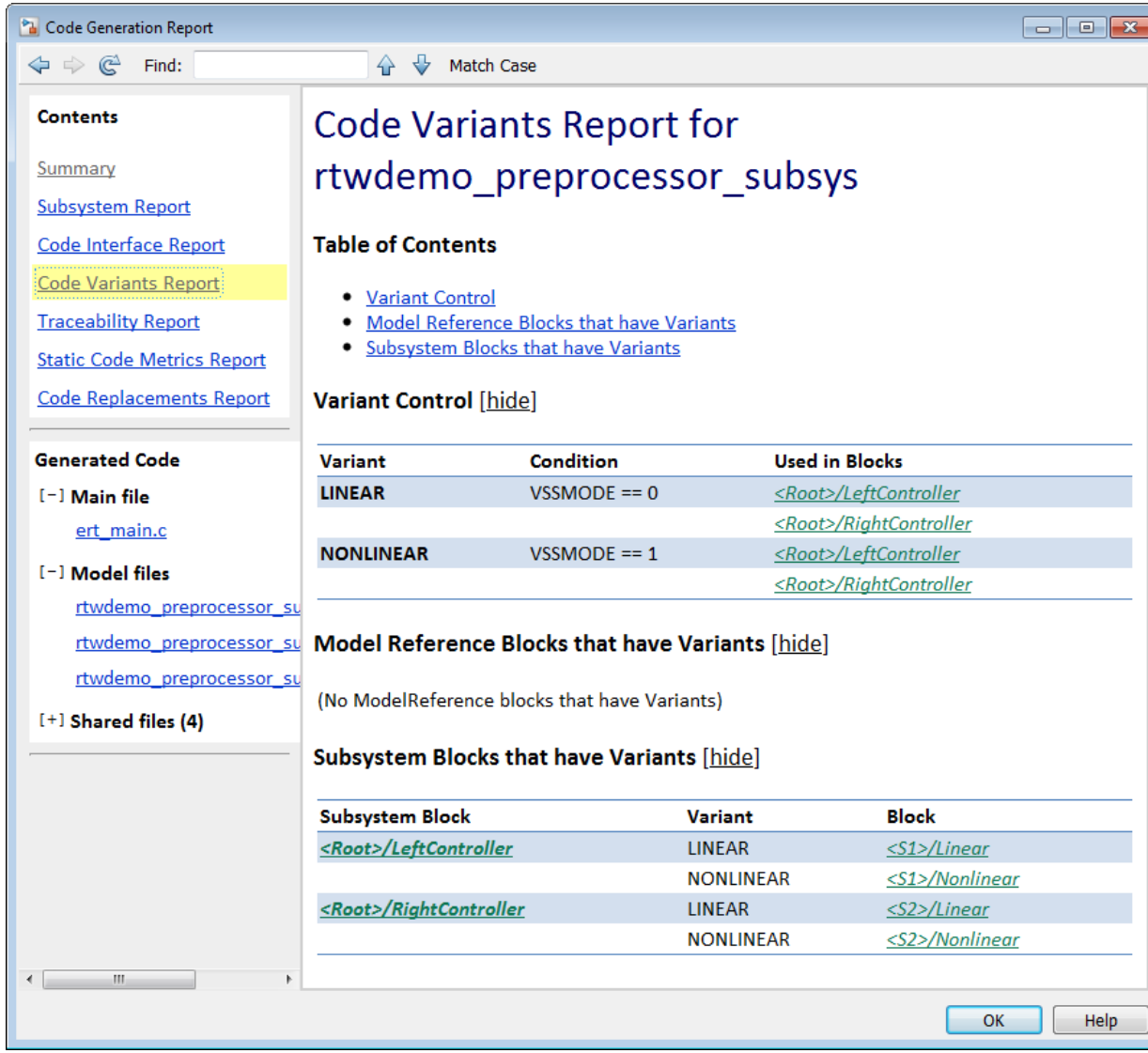
Step 4: Review Generated Code

The code generation report contains a section dedicated to the subsystems that have variants controlled by preprocessor conditionals.

1 To open the Code Generation Report click **Code > C/C++ Code > Code Generation Report > Open Model Report**.



2 Select the **Code Variant Report** from the left.



In this example, the generated code includes references to the `Simulink.Variant` objects `LINEAR` and `NONLINEAR`. The code also includes the definitions of macros corresponding to those variants. The definitions depend on the value of `VSSMODE`,

which is supplied in an external header file `rtwdemo_importedmacros.h`. The active variant is determined by using preprocessor conditionals (`#if`) on the macros (`#define`) `LINEAR` and `NONLINEAR`.

3 Select the `rtwdemo_preprocessor_subsys_types.h` file from the left.

This file contains the definitions of macros `LINEAR` and `NONLINEAR`.

```
#ifndef LINEAR
 #define LINEAR      (VSSMODE == 0)
#endif

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

4 Select the `rtwdemo_preprocessor_subsys.c` file from the left.

In this file, calls to the step and initialization functions of each variant are conditionally compiled.

```
 /* Outputs for Atomic SubSystem: '<Root>/LeftController' */
#if LINEAR
 /* Output and update for atomic system: '<S1>/Linear' */
 ...
#elif NONLINEAR
 /* Output and update for atomic system: '<S1>/Nonlinear' */
 ...
#endif
```

Limitations

- When you are generating code for Model Variants blocks and Variant Subsystem blocks, the blocks cannot have:
  - Mass matrices
  - Function call ports
  - Outports with constant sample time
  - Simscape blocks
- The Model Variants block and its referenced models must have the same number of inports and outports.

- The port numbers and names for each active child subsystem must belong to a subset of the port numbers and names of the parent Variant Subsystem block.

Related Examples

- "Define, Configure, and Activate Variants" (Simulink)
- "Variant Subsystems" (Simulink)
- "Model Reference Variants" (Simulink)

More About

- "What Are Variants and When to Use Them" (Simulink)
- "Introduction to Variant Controls" (Simulink)

Generate Preprocessor Conditionals for Variant Systems

In this section...

Define Variant Controls

For variant systems, conditional expressions called variant controls determine which variant choice is active. This example shows how to define variant controls for generating code.

1. Open the Model Explorer. Select the **base workspace**.
2. A variant control can be a condition expression, a Simulink.Variant class (Simulink) object specifying a condition expression or a Simulink.Parameter object. In the Model Explorer, select **Add** > **Simulink Parameter**. Specify a name for the new parameter.
3. Use the function `Simulink.VariantManager.findVariantControlVars` to find and convert MATLAB variables used in variant control expressions into `Simulink.Parameter` objects. For an example, see "Convert Variant Control Variables into Simulink.Parameter Objects" (Simulink).
4. On the `Simulink.Parameter` property dialog box, specify the **Value** and **Data type**.
5. Select one of these **Storage class** values.
   - `Define`
   - `ImportedDefine(Custom)`
   - `CompilerFlag(Custom)`
   - A storage class created using the Custom Storage Class Designer. Your storage class must have the **Data initialization** parameter set to `Macro` and the **Data scope** parameter set to `Imported`. See "Use Custom Storage Class Designer" on page 23-35 for more information.
6. Specify the value of the variant control. If the storage class is `ImportedDefine(Custom)`, do the following:

a Specify the **Header File** parameter as an external header file in the Custom Attributes section of the `Simulink.Parameter` property dialog box.

b Enter the values of the variant controls in the external header file.

Note: The generated code refers to a variant control as a user-defined macro. The generated code does not contain the value of the macro. The value of the variant control determines the active variant in the compiled code.

If the variant control is a `CompilerFlag` custom storage class, the value of the variant control is set at compile time. Use the **Configuration Parameters > Code Generation > Custom Code > Additional build information > Defines** parameter to add a list of variant controls (macro definitions) to the compiler command line. For example, for variant control `VSSMODE`, in the text field for the **Defines** parameter, enter:

```
-DVSSMODE=1
```

If you want to modify the value of the variant control after generating a makefile, use a makefile option when compiling your code. For example, at a command line outside of MATLAB, enter:

```
makecommand -f model.mk DEFINES_CUSTOM="-DVSSMODE=1"
```

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

Configure Model for Generating Preprocessor Conditional Directives

1 Open the Configuration Parameters dialog box.

2 Select the **Code Generation** pane, and set **System target file** as `ert.tlc`.

3 In the **Report** pane, select **Create code generation report**.

4 On the **All Parameters** tab, clear "Ignore custom storage classes" (Simulink Coder). In order to generate preprocessor conditionals, you must use custom storage classes.

5. In the Model Variants block parameter dialog box, select the **Generate preprocessor conditionals** parameter option. In the Variant Subsystem, Variant Source, or Variant Sink block parameter dialog boxes, select the **Analyze all choices during update diagram and generate preprocessor conditionals** option.
6. In both cases, clear the option to **Override variant conditions and use following variant**.
7. Generate code.

Special Considerations for Generating Preprocessor Conditionals

- The code generation process checks that the inports and outports of a Model Variants block are identical (same port numbers and names) to the corresponding inports and outports of its variants. The build process for simulation does not make this check. Therefore, if your variant block contains mismatched inports or outports, the code generation process issues an error.
- The port numbers and names for each child variant subsystem must belong to a subset of the port numbers and names of the parent Variant Subsystem block.
- The code generation process checks that there is at least one active variant by using the variant control values stored in the base workspace. The variant control that evaluates to `true` becomes the active variant. If none of the variant controls evaluates to `true`, the default variant, if specified, becomes the active variant. The code generation process issues an error if an active variant does not exist.
- Implement the condition expressions of the variant objects such that only one evaluates to `true`. The generated code includes a test of the variant objects to determine that there is only one active variant. If this test fails, your code will not compile.
- If you comment out child subsystems listed in the **Variant Choices** table in the Variant Subsystem block parameter dialog box, the code generator does not generate code for the commented out subsystems.
- If the sample time for a default variant differs from that of the other variant choices, the `#else` preprocessor conditional is not generated for the default variant. Instead, an `#if !(`*`<variant conditions>`*`)` is generated.
- For Variant Subsystems, the `model_private.h` file contains conditional parameter definitions. For example, if the value of a Constant block is a `Simulink.Parameter` with an `ImportedDefine` custom storage class, and the Constant block is in a Variant Subsystem, the conditional definition of the `Simulink.Parameter` is in the `model_private.h` file.

Related Examples

- “Create Variant Controls Programmatically” (Simulink)
- “Working with Variant Choices” (Simulink)

Represent Variant Source and Sink Blocks in Generated Code

In this section...

You can use Variant Source and Variant Sink blocks to perceive multiple implementations of a model in a single, unified block diagram. Each implementation depends on conditions that you set for Variant Source and Variant Sink blocks. Simulink propagates these conditions to upstream and downstream blocks including root input and root output ports.

You can generate:

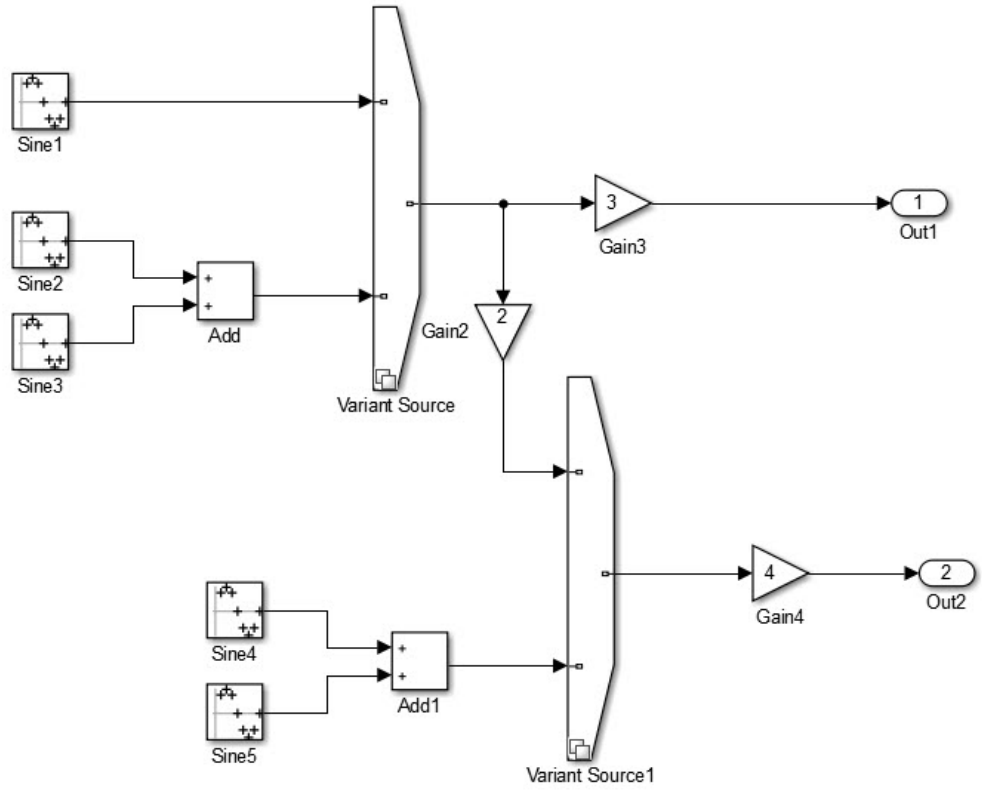
- Code from a Simulink model containing Variant Sink and Variant Source blocks.
- Code that contains preprocessor conditionals that control the activation of each variant choice.
- Preprocessor conditionals that allow for no active variant choice.

Represent Variant Source and Variant Sink blocks in Simulink

This example shows how Variant Source blocks make model elements conditional.

1. From the Simulink Block Library, add 1 Sine Wave Function block, two Add blocks, three Gain blocks, two Outports, and two Variant Source blocks into a new model.

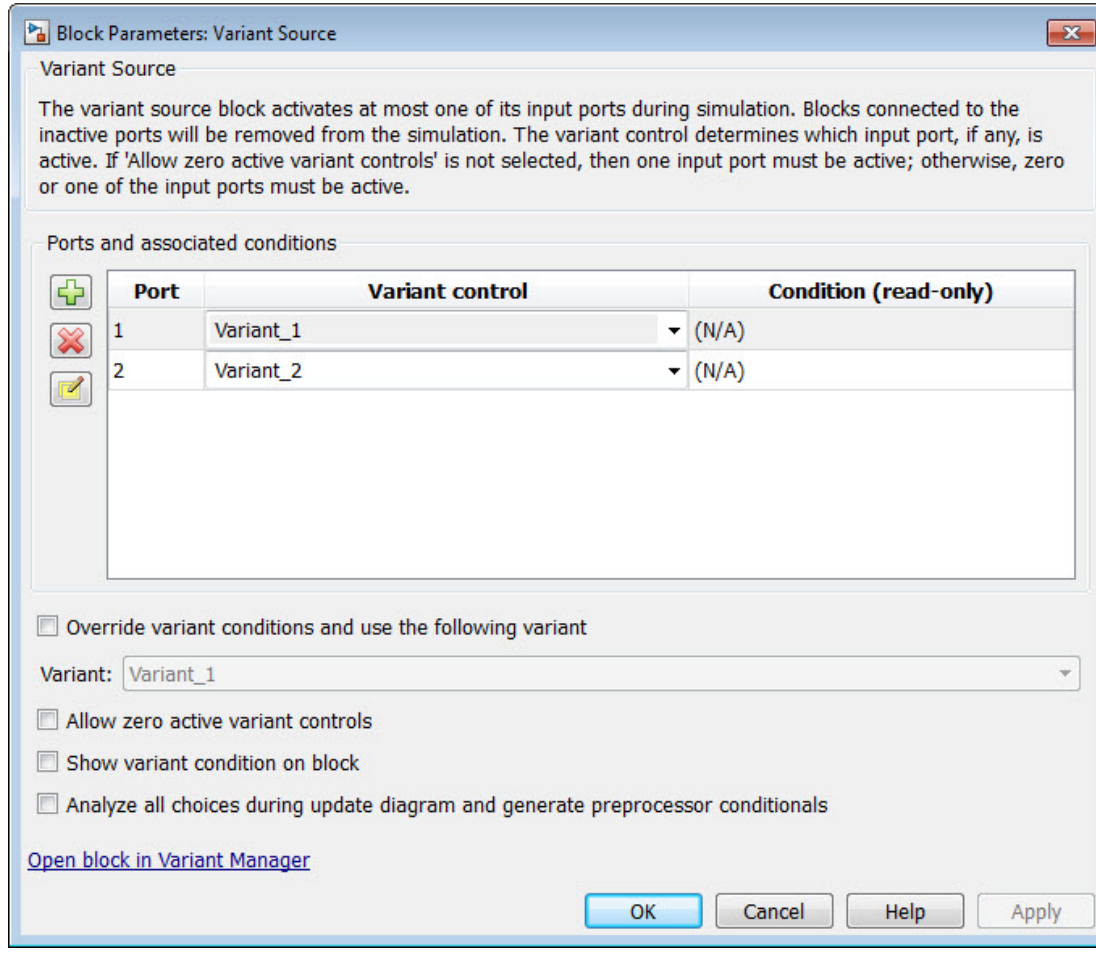
2. Open the Sine Wave Function block. For the **Sine type** parameter, select `Sample based`. For the **Time (t)** parameter, select `Use simulation time`. For the **Sample time** parameter, insert a value of `0.2`.
3. Make four copies of the Sine Wave Function block.
4. Connect and name the blocks as shown.



5 Insert values of `2`, `3`, and `4` in the `Gain2`, `Gain3`, and `Gain4` blocks, respectively.

6 Give the model the name `inline_variants_example`.

7 Open the Block Parameters dialog box for `Variant Source`.



8 In the **Variant control** column, for Port 1, replace `Variant_1` with `V==1`. For Port 2, replace `Variant_2` with `V==2`.

9 Open the Block Parameters dialog box for `Variant Source1`.

10 In the **Variant control** column, replace `Variant_1` with `W==1`. For Port 2, replace `Variant_2` with `W==2`.

11 In the MATLAB Command Window, to create `Simulink.Parameter` for the variant control variables `V` and `W`, use these commands:

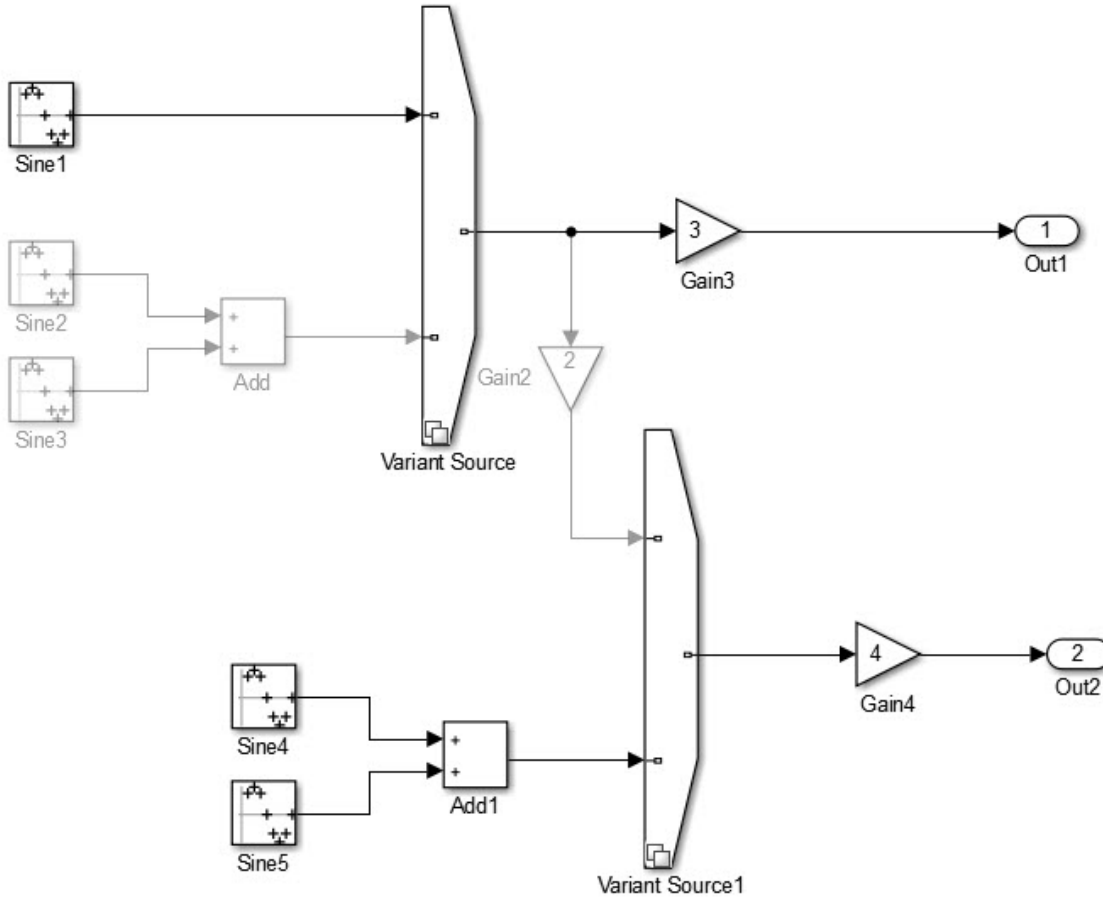
```
V = Simulink.Parameter;
V.Value = 1;
V.DataType='int32';
V.CoderInfo.StorageClass = 'custom';
V.CoderInfo.CustomStorageClass = 'Define';
V.CoderInfo.CustomAttributes.HeaderFile='inline_importedmacro.h'

W = Simulink.Parameter;
W.Value = 2;
W.DataType='int32';
W.CoderInfo.StorageClass = 'custom';
W.CoderInfo.CustomStorageClass = 'Define';
W.CoderInfo.CustomAttributes.HeaderFile='inline_importedmacro.h'
```

In this example, the variant control variables are `Simulink.Parameter` objects. For code generation, if you use `Simulink.Variant` objects to specify variant controls, use `Simulink.Parameter` objects to specify their conditions. If you use scalar variant control variables to simulate the model, you can convert those variables into `Simulink.Parameter` objects. See "Convert Variant Control Variables into Simulink.Parameter Objects" (Simulink).

Variant control variables defined as `Simulink.Parameter` objects can have one of these storage classes:

- `Define` with header file specified
- `ImportedDefine` with header file specified
- `CompilerFlag`
- `SystemConstant (AUTOSAR)`
- User-defined custom storage class that defines data as a macro in a specified header file

12 Simulate the model.



Input port 1 is the active choice for `Variant Source` because the value of variant control variable `V` is `1`. Input port 2 is the active choice for `Variant Source1` because the value of variant control variable `W` is 2. The inactive choices are removed from execution, and their paths are grayed-out in the diagram.

Specify Conditions That Control Variant Choice Selection

You can generate code in which each variant choice is enclosed within C preprocessor conditionals `#if` and `#endif`. The compiler chooses the active variant at compile time and the preprocessor conditionals determine which sections of the code to execute.

1. In the Simulink editor, select **Simulation** > **Model Configuration Parameters**.
2. Select the **Code Generation** pane, and set **System target file** to `ert.tlc`.
3. In the **Report** pane, select **Create code generation report**.
4. In your model, open the block parameters dialog box for `Variant Source`.
5. Select the **Analyze all choices during update diagram and generate preprocessor conditionals** parameter. During an update diagram or simulation, when you select this parameter, Simulink analyzes all variant choices. This analysis provides early validation of the code generation readiness of variant choices. During code generation, when you select this parameter, the code generator generates preprocessor conditionals that control the activation of each variant choice.
6. Clear the **Override variant conditions and use the following variant** parameter.
7. Clear the **Allow zero active variant controls** parameter.
8. Open the Block Parameters dialog box for `Variant Source 1`. Repeat steps 5 through 7.
9. Build the model. When code generation is complete, the Code Generation Report is displayed.

Review the Generated Code

1. In the code generation report, select the `inline_variants_example.c` file.
2. In the `inline_variants_example.c` file, calls to the `inline_variants_example_step` function and the `inline_variants_example_initialize` functions are conditionally compiled as shown:

```
/* Model step function */
void inline_variants_example_step(void)
{
  real_T rtb_Sine4;
  real_T rtb_VariantMerge_For_Variant_So;
```

```
  /* Sin: '<Root>/Sine1' */
#if V == 1

  rtb_Sine4 = sin((real_T)inline_variants_example_DW.counter * 2.0 *
                  3.1415926535897931 / 10.0);

#endif                                 /* V == 1 */

  /* End of Sin: '<Root>/Sine1' */

  /* Sin: '<Root>/Sine2' incorporates:
   *  Sin: '<Root>/Sine3'
   *  Sum: '<Root>/Add'
   */
#if V == 2

  rtb_Sine4 = sin((real_T)inline_variants_example_DW.counter_i * 2.0 *
                  3.1415926535897931 / 10.0) + sin((real_T)
    inline_variants_example_DW.counter_f * 2.0 * 3.1415926535897931 / 10.0);

#endif                                 /* V == 2 */

  /* End of Sin: '<Root>/Sine2' */

  /* Outport: '<Root>/Out1' incorporates:
   *  Gain: '<Root>/Gain3'
   */
  inline_variants_example_Y.Out1 = 3.0 * rtb_Sine4;

  /* Gain: '<Root>/Gain2' */
#if W == 1

  rtb_VariantMerge_For_Variant_So = 2.0 * rtb_Sine4;

#endif                                 /* W == 1 */

  /* End of Gain: '<Root>/Gain2' */

  /* Sin: '<Root>/Sine4' incorporates:
   *  Sin: '<Root>/Sine5'
   *  Sum: '<Root>/Add1'
   */
#if W == 2
```

```
  rtb_VariantMerge_For_Variant_So = sin((real_T)
    inline_variants_example_DW.counter_fe * 2.0 * 3.1415926535897931 / 10.0) *
    2.0 + sin((real_T)inline_variants_example_DW.counter_e * 2.0 *
              3.1415926535897931 / 10.0);

#endif                                 /* W == 2 */

  /* End of Sin: '<Root>/Sine4' */

/* Outport: '<Root>/Out2' incorporates:
   *  Gain: '<Root>/Gain4'
   */
  inline_variants_example_Y.Out2 = inline_variants_example_P.Gain4_Gain *
    rtb_VariantMerge_For_Variant_So;
...
}
```

The variables `rtb_Sine4` and `rtb_VariantMerge_For_Variant_So` hold the input values to the Variant Source blocks. Notice that the code for these variables is conditional. The variables `inline_variants_example_Y.Out1` and `inline_variants_example_Y.Out2` hold the output values of the Variant Source blocks. Notice that the code for these variables is not conditional.

Generate Code with Zero Active Variant Controls

You can generate code in which blocks connected to the input and the output of a Variant Source block are conditional.

1. For `Variant Source`, open the Block Parameters dialog box. Select the parameter **Allow zero active variant controls**.
2. For `Variant Source 1`, open the Block Parameters dialog box. Select the parameter **Allow zero active variant controls**.

When you select **Allow zero active variant controls** parameter, you can generate code for a model containing Variant Source and Variant Sink blocks even when you specify a value for a variant control variable that does not allow for an active variant. Choosing a value for a variant control variable that does not allow for an active variant and not selecting the **Allow zero active variant controls** parameter, produces an error.

Generate code for `inline_variants_example`. Notice in the `inline_variants_example.c` file, that the code for the variables

`inline_variants_example_Y.Out1` and `inline_variants_example_Y.Out2` is conditional.

```
/* Model step function */
void inline_variants_example_step(void)
{
 ...
 #if V == 1 || V == 2

  inline_variants_example_Y.Out1 = 3.0 * rtb_Sine4;

#endif                                 /* V == 1 || V == 2 */

 ...
#if (V == 1 && W == 1) || (V == 2 && W == 1) || W == 2

  inline_variants_example_Y.Out2 = 4.0 * rtb_VariantMerge_For_Variant_So;

#endif                      /* (V == 1 && W == 1) || (V == 2 && W == 1) || W == 2 */

 ...
```

Global Data Guarding Limitation

For external ports and most DWork vectors, signals, and states, preprocessor conditionals (`#if` and `#endif`) surround global data variable declarations. For models in which you enable C API code for global block output signals, global block parameters, and discrete and continuous states, preprocessor conditionals do not surround global data variable declarations. For information on the C API, see "Exchange Data Between Generated and External Code Using C API" (Simulink Coder).

State Logging Limitation

There are some rare cases in which preprocessor conditionals do not surround global data structures that contain state variable declarations. For models that contain Variant Source blocks or Variant Sink blocks and also contain blocks that maintain state information, such as Unit Delay blocks, the exclusion of preprocessor conditionals surrounding state variable declarations can lead to a mismatch between simulation and code generation results.

For example, suppose that a model has a Variant Source block with four variant choices. One of these choices contains blocks with state information. If you simulate

the model with the active variant that is other than the variant choice that contains state information, there is no logged state data. In the *model*`.h` file, the generated code still initializes these global state variables to `0` because `#if` and `#endif` guards do not surround the state variable declarations. If you create a *model*`.mat` file from a *model*`.exe` file, and compare it to the simulation output, the results do not match. For this example, the simulation output is empty because there is no logged state data. The *model*`.mat` file contains multiple values of `0`.

If the active variant is the variant choice containing state information, the results do match.

Related Examples

- "Define and Configure Variant Sources and Sinks" (Simulink)
- "Variant Condition Propagation with Variant Sources and Sinks" (Simulink)
- "Introduction to Variant Controls" (Simulink)

Configure Dimension Variants for S-Function Blocks

To configure symbolic dimensions for S-function blocks, you can use the following C/C ++ functions. You can configure S-functions to support forward propagation, backward propagation, or forward and backward propagation of symbolic dimensions during simulation.

Many of these functions return the variable `SymbDimsId`. A `SymbDimsId` is a unique integer value. This value corresponds to each symbolic dimension specification that you create or is the result of a mathematical operation that you perform with symbolic dimensions.

Note: If you are writing an S-function with symbolic dimensions, you can not use the `%roll` directive. You must write an explicit loop.

C/C++ S-Functions	Purpose
ssSetSymbolicDimsSupport	Specify whether or not an S-function supports symbolic dimensions.
mdlSetInputPortSymbolicDimensions	Specify the symbolic dimensions of an input port and how those dimensions propagate forward.
mdlSetOutputPortSymbolicDimensions	Specify the symbolic dimensions of an output port and how those dimensions propagate backward.
ssRegisterSymbolicDimsExpr	Create a `SymbDimsId` from an expression string (`aExpr`). The expression string must form a valid syntax in C.
ssRegisterSymbolicDims	Create a `SymbDimsId` from a vector of `SymbDimsId`s.
ssRegisterSymbolicDimsString	Create a `SymbDimsId` from an identifier string (`aString`).
ssRegisterSymbolicDimsIntValue	Create a `SymbDimsId` from an integer value (`aIntValue`)

C/C++ S-Functions	Purpose
ssRegisterSymbolicDimsPlus	Create a `SymbDimsId` by adding two symbolic dimensions.
ssRegisterSymbolicDimsMinus	Create a `SymbDimsId` by subtracting two symbolic dimensions.
ssRegisterSymbolicDimsMultiply	Create a `SymbDimsId` by multiplying two symbolic dimensions.
ssRegisterSymbolicDimsDivide	Create a `SymbDimsId` by dividing two symbolic dimensions.
ssGetNumSymbolicDims	Get the number of dimensions for a `SymbDimsId`.
ssGetSymbolicDim	Get a `SymbDimsId` from a vector of `SymbDimsId`s.
ssSetInputPortSymbolicDimsId	Set the precompiled `SymbDimsId` of an input port. You can call this function from inside the `mdlInitializeSizes` function.
ssGetCompInputPortSymbolicDimsId	Get the compiled `SymbDimsId` of an input port.
ssSetCompInputPortSymbolicDimsId	Set the compiled `SymbDimsId` of an input port.
ssSetOutputPortSymbolicDimsId	Set the precompiled `SymbDimsId` of an output port. You can call this function from inside the `mdlInitializeSizes` function.
ssGetCompOutputPortSymbolicDimsId	Get the compiled `SymbDimsId` of an output port.
ssSetCompOutputPortSymbolicDimsId	Set the compiled `SymbDimsId` of an output port.
ssSetCompDWorkSymbolicDimsId	Set the compiled `SymbDimsId` of an index of a block's data type work (`DWork`) vector.

S-Function That Supports Forward Propagation of Symbolic Dimensions

This S-function subtracts the symbolic dimension B from a symbolic input dimension. It does not support backward propagation of symbolic dimensions because the compiled symbolic dimensions of the input port are not set. Symbolic dimensions are set for the output port, so forward propagation occurs.

```
static void mdlInitializeSizes(SimStruct *S)
{
     // Enable symbolic dimensions for the s-function.
     ssSetSymbolicDimsSupport(S, true);
}

#if defined(MATLAB_MEX_FILE)
#define MDL_SET_INPUT_PORT_SYMBOLIC_DIMENSIONS
static void mdlSetInputPortSymbolicDimensions(SimStruct* S,
   int_T portIndex, SymbDimsId symbDimsId)
{
    assert(0 == portIndex);
    // Set the compiled input symbolic dimension.
    ssSetCompInputPortSymbolicDimsId(S,  portIndex, symbDimsId);
    // Register “B” and get its symbolic dimensions id.
    const SymbDimsId symbolIdForB = ssRegisterSymbolicDimsString(S, "B");
   // Subtract “B” from the input symbolic dimension.
    const SymbDimsId outputDimsId =
        ssRegisterSymbolicDimsMinus(S, symbDimsId, symbolIdForB);
   //Set the resulting symbolic dimensions id as the output.
    ssSetCompOutputPortSymbolicDimsId(S, portIndex, outputDimsId);
}
#endif

#if defined(MATLAB_MEX_FILE)
#define MDL_SET_OUTPUT_PORT_SYMBOLIC_DIMENSIONS
static void mdlSetOutputPortSymbolicDimensions(SimStruct *S,
   int_T portIndex, SymbDimsId symbDimsId)
{
    assert(0 == portIndex);
   // The input dimensions are not set, so this S-function only
 // supports forward propagation.
    ssSetCompOutputPortSymbolicDimsId(S, portIndex, symbDimsId);
}
#endif
```

S-Function That Supports Forward and Backward Propagation of Symbolic Dimensions

This S-function transposes two symbolic dimensions. It supports forward and backward propagation of symbolic dimensions because the compiled symbolic dimension of both the input and output ports are set.

```
static void mdlInitializeSizes(SimStruct *S)
{
     // Enable symbolic dimensions for the s-function.
     ssSetSymbolicDimsSupport(S, true);
}

#if defined(MATLAB_MEX_FILE)
#define MDL_SET_INPUT_PORT_SYMBOLIC_DIMENSIONS
static void mdlSetInputPortSymbolicDimensions(SimStruct* S,
   int_T portIndex, SymbDimsId symbDimsId)
{
    assert(0 == portIndex);

    ssSetCompInputPortSymbolicDimsId(S,  portIndex, symbDimsId);

    assert(2U == ssGetNumSymbolicDims(S, symbDimsId));

    if (SL_INHERIT ==
        ssGetCompOutputPortSymbolicDimsId(S, portIndex)) {

        const SymbDimsId idVec[] = {
            ssGetSymbolicDim(S, symbDimsId, 1),
            ssGetSymbolicDim(S, symbDimsId, 0)};
        // Register the transposed dimensions.
       // Set the output symbolic dimension to the resulting id.
  const SymbDimsId outputDimsId =
            ssRegisterSymbolicDims(S, idVec, 2U);

        ssSetCompOutputPortSymbolicDimsId(S, portIndex,
           outputDimsId);
    }
}
#endif

#if defined(MATLAB_MEX_FILE)
#define MDL_SET_OUTPUT_PORT_SYMBOLIC_DIMENSIONS
static void mdlSetOutputPortSymbolicDimensions(SimStruct *S,
```

```
    int_T portIndex, SymbDimsId symbDimsId)
{
    assert(0 == portIndex);
    ssSetCompOutputPortSymbolicDimsId(S, portIndex, symbDimsId);

    assert(2U == ssGetNumSymbolicDims(S, symbDimsId));

    if (SL_INHERIT ==
        ssGetCompInputPortSymbolicDimsId(S, portIndex)) {

        const SymbDimsId idVec[] = {
            ssGetSymbolicDim(S, symbDimsId, 1),
            ssGetSymbolicDim(S, symbDimsId, 0)};
        const SymbDimsId inputDimsId =
            ssRegisterSymbolicDims(S, idVec, 2U);
        // Register the transposed dimensions.
       // Set the input symbolic dimension to the resulting id.
        ssSetCompInputPortSymbolicDimsId(S, portIndex, inputDimsId);
    }
}
#endif
```

Related Examples

- “Implement Dimension Variants for Array Sizes in Generated Code” on page 14-2

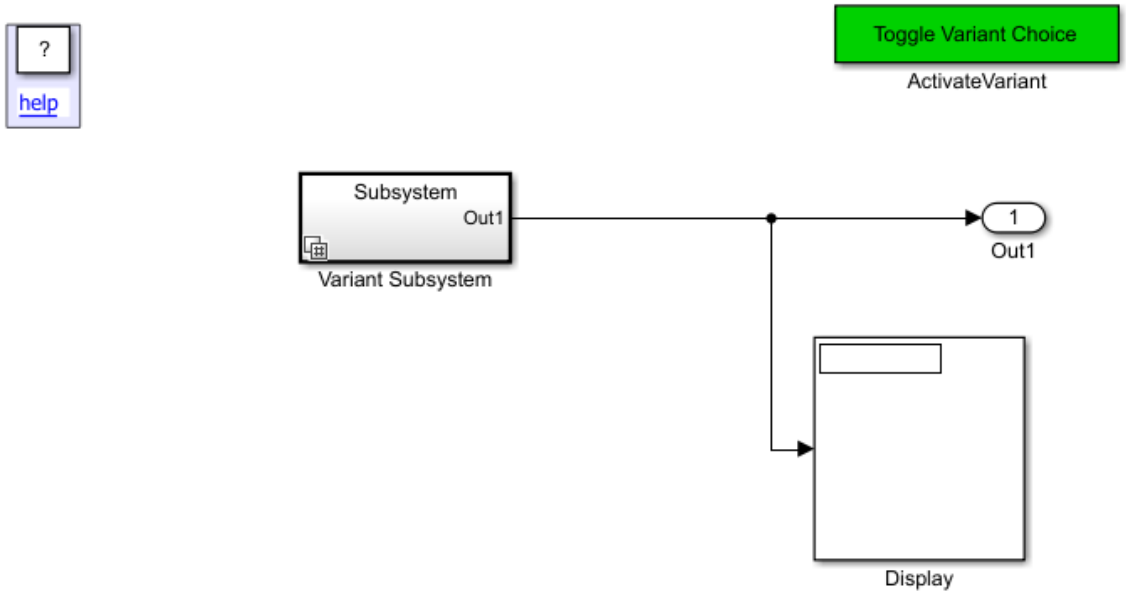
Generate Code for Variant Subsystem with Child Subsystems of Different Output Signal Dimensions

In this section...

This example shows how to use symbolic dimensions to generate code with preprocessor conditionals for a variant subsystem consisting of child subsystems of different output signal dimensions. The value of the variant control variable determines the active variant choice and the output signal dimensions. By changing the value of the variant control variable, you change the active variant and the output signal dimensions in the generated code.

Example Model

The model `slexVariantSymbolicDims` contains a Variant Subsystem consisting of the child subsystems `Subsystem` and `Subsystem1`. When the variant control variable `Var` has a value of `1`, `Subsystem` is the active variant. When `Var` has a value of `2`, `Subsystem1` is the active variant.



Simulate Model

To generate code with preprocessor conditionals, the output signal dimensions of the child subsystems must be the same during simulation. In this example, double-clicking the subsystem `Activate Variant Choice` changes the active variant and the output signal dimension. When `Var` equals `1`, the output signal dimension of each child subsystem is `5`. When `Var` equals `2`, the output signal dimension of each child subsystem is `6`.

1. Open the example model slexVariantSymbolicDims.
2. From the **Display** > **Signals & Ports** menu, select **Signal Dimensions**.
3. Open the Variant Subsystem Block Parameters dialog box. The **Analyze all choices during update diagram and generate preprocessor conditionals** parameter is selected.
4. Open `Subsystem`. In the Constant Block Parameters dialog box, the **Constant value** parameter is `P1`.
5. Open `Subsystem1`. In the Constant Block Parameters dialog box, the **Constant value** parameter is `P2`.

6. Open the base workspace. The `Simulink.Parameters` `P1` and `P2` are arrays with dimensions `'[1,A]'`. The `Simulink.Parameter` `A` has a value of `5`. `Var` has a value of `1`.
7. Simulate the model. `Subsystem` is the active variant with an output signal dimension of `5`.
8. Double-click the masked subsystem `ActivateVariant`.
9. In the base workspace, `Var` has a value of `2`. `P1` and `P2` have a dimension of `6`. `A` has a value of `6`.
10. Simulate the model. `Subsystem1` is the active variant with an output signal dimension of `6`.

In the base workspace, `A` has a **Storage class** of `ImportedDefine(Custom)`. To use a `Simulink.Parameter` object for dimension specification, it must have one of these storage classes:

- `Define` or `ImportedDefine` with header file specified
- `CompilerFlag`
- User-defined custom storage class that defines data as a macro in a specified header file

In the base workspace, `P1` and `P2` have a storage class of `ImportedExtern`. A `Simulink.Parameter` object that uses a `Simulink.Parameter` for symbolic dimension specification must have a storage class of either `ImportedExtern` or `ImportedExternPointer`.

Generate Code

1. Open the header file slexVariantSymbolicDims_variant_defines.h. The definition of `A` is conditional upon the value of `Var`.

```
/* Copyright 2016 The MathWorks, Inc. */
// To select variant choice during compile, define Var at compile time,

#ifndef Var
#define Var 1
#endif

#if Var == 1
#define A 5
```

```
#elif Var == 2
#define A 6
#else
#error "Variant control variable, Var, must be defined as 1 or 2"
#endif
```

2. Generate code.
3. Open the `slexVariantSymbolicDims.h` file. The output dimension size is `A`.

```
/* External outputs (root outports fed by signals with auto storage) */
typedef struct {
  int32_T Out1[A];                     /* '<Root>/Out1' */
} ExternalOutputs_slexVariantSymb;
```

4. Open the `slexVariantSymbolicDims.c` file. If `Var` equals `1`, `P1` has five values. If `Var` equals `2`, `P2` has six values. In the Configuration Parameters dialog box, on the **Code Generation** > **Custom Code** pane, the **Source file** parameter contains this code.

```
/* user code (top of source file) */
#if Var == 1

int32_T P1[] = { 5, 5, 5, 5, 5 };

#elif Var == 2

int32_T P2[] = { 6, 6, 6, 6, 6, 6 };

#endif
```

Preprocessor conditionals control the size of `A` and which array, `P1` or `P2`, is active in the generated code. By changing the value of `Var`, you can change the size of `A` and the active array.

Related Examples

- "Implement Dimension Variants for Array Sizes in Generated Code" on page 14-2
- "Represent Subsystem and Model Variants in Generated Code" on page 14-21

15

Timers in Simulink Coder

Absolute and Elapsed Time Computation

In this section...

About Timers

Certain blocks require the value of either *absolute* time (that is, the time from the start of program execution to the present time) or *elapsed* time (for example, the time elapsed between two trigger events). Targets that support the real-time model (`rtModel`) data structure provide efficient time computation services to blocks that request absolute or elapsed time. Absolute and elapsed timer features include

- Timers are implemented as unsigned integers in generated code.
- In multirate models, at most one timer is allocated per rate. If no blocks executing at a given rate require a timer, a timer is not allocated to that rate. This minimizes memory allocated for timers and significantly reduces overhead involved in maintaining timers.
- Allocation of elapsed time counters for use of blocks within triggered subsystems is minimized, further reducing memory usage and overhead.
- S-function and TLC APIs let your S-functions access timers, in simulation and code generation.
- The word size of the timers is determined by a user-specified maximum counter value, **Application lifespan (days)** (Simulink). If you specify this value, timers will not overflow. For more information, see "Control Memory Allocation for Time Counters" (Simulink Coder).

See "Absolute Time Limitations" (Simulink Coder) for more information about absolute time and the restrictions that it imposes.

Timers for Periodic and Asynchronous Tasks

Timing services provided for blocks execute within *periodic* tasks (that is, tasks running at the model base rate or subrates).

The code generator also provides timer support for blocks whose execution is *asynchronous* with respect to the periodic timing source of the model. See the following topics:

- "Timers in Asynchronous Tasks" on page 17-44
- "Create a Customized Asynchronous Library" on page 17-47

Allocation of Timers

If you create or maintain an S-Function block that requires absolute or elapsed time data, it must register the requirement (see "Access Timers Programmatically" on page 15-5). In multirate models, timers are allocated on a per-rate basis. For example, consider a model structured as follows:

- There are three rates, A, B, and C, in the model.
- No blocks running at rate B require absolute or elapsed time.
- Two blocks running at rate C register a requirement for absolute time.
- One block running at rate A registers a requirement for absolute time.

In this case, two timers are generated, running at rates A and C respectively. The timing engine updates the timers as the tasks associated with rates A and C execute. Blocks executing at rates A and C obtain time data from the timers associated with rates A and C.

Integer Timers in Generated Code

In the generated code, timers for absolute and elapsed time are implemented as unsigned integers. The default size is 64 bits. This is the amount of memory allocated for a timer if you specify a value of `inf` for the **Application lifespan (days)** (Simulink) parameter. For an application with a sample rate of 1000 MHz, a 64-bit counter will not overflow for more than 500 years. See "Timers in Asynchronous Tasks" on page 17-44 and "Control Memory Allocation for Time Counters" on page 53-11 for more information.

Elapsed Time Counters in Triggered Subsystems

Some blocks, such as the Discrete-Time Integrator block, perform computations requiring the elapsed time (delta T) since the previous block execution. Blocks requiring elapsed time data must register the requirement (see “Access Timers Programmatically” on page 15-5). A triggered subsystem then allocates and maintains a single elapsed time counter if required. This timer functions at the subsystem level, not at the individual block level. The timer is generated if the triggered subsystem (or a unconditionally executed subsystem within the triggered subsystem) contains one or more blocks requiring elapsed time data.

Note: If you are using simplified initialization mode, elapsed time is reset on first execution after becoming enabled, whether or not the subsystem is configured to reset on enable. For more information, see “Underspecified initialization detection” (Simulink).

More About

- “Access Timers Programmatically” (Simulink Coder)
- “Generate Code for an Elapsed Time Counter” (Simulink Coder)
- “Optimize Memory Usage for Time Counters” (Simulink Coder)
- “Absolute Time Limitations” (Simulink Coder)

Access Timers Programmatically

In this section...

About Timer APIs

This topic describes APIs that let your S-functions take advantage of the efficiencies offered by absolute and elapsed timers. SimStruct macros are provided for use in simulation, and TLC functions are provided for inlined code generation. Note that

- To generate and use the new timers as described above, your S-functions must register the need to use an absolute or elapsed timer by calling `ssSetNeedAbsoluteTime` or `ssSetNeedElapseTime` in `mdlInitializeSampleTime`.
- Existing S-functions that read absolute time but do not register by using these macros continue to operate as expected, but generate less efficient code.

C API for S-Functions

The SimStruct macros described in this topic provide access to absolute and elapsed timers for S-functions during simulation.

In the functions below, the `SimStruct *S` argument is a pointer to the `simstruct` of the calling S-function.

- `void ssSetNeedAbsoluteTime(SimStruct *S, boolean b)`: if `b` is `TRUE`, registers that the calling S-function requires absolute time data, and allocates an absolute time counter for the rate at which the S-function executes (if such a counter has not already been allocated).
- `int ssGetNeedAbsoluteTime(SimStruct *S)`: returns 1 if the S-function has registered that it requires absolute time.
- `double ssGetTaskTime(SimStruct *S, tid)`: read absolute time for a given task with task identifier `tid`. `ssGetTaskTime` operates transparently, regardless of whether or not you use the new timer features. `ssGetTaskTime` is documented in the SimStruct Functions chapter of the Simulink documentation.

- `void ssSetNeedElapseTime(SimStruct *S, boolean b)`: if `b` is `TRUE`, registers that the calling S-function requires elapsed time data, and allocates an elapsed time counter for the triggered subsystem in which the S-function executes (if such a counter has not already been allocated). See also "Elapsed Time Counters in Triggered Subsystems" on page 15-4.
- `int ssGetNeedElapseTime(SimStruct *S)`: returns 1 if the S-function has registered that it requires elapsed time.
- `void ssGetElapseTime(SimStruct *S, (double *)elapseTime)`: returns, to the location pointed to by `elapseTime`, the value (as a `double`) of the elapsed time counter associated with the S-function.
- `void ssGetElapseTimeCounterDtype(SimStruct *S, (int *)dtype)`: returns the data type of the elapsed time counter associated with the S-function to the location pointed to by `dtype`. This function is intended for use with the `ssGetElapseTimeCounter` function (see below).
- `void ssGetElapseResolution(SimStruct *S, (double *)resolution)`: returns the resolution (that is, the sample time) of the elapsed time counter associated with the S-function to the location pointed to by `resolution`. This function is intended for use with the `ssGetElapseTimeCounter` function (see below).
- `void ssGetElapseTimeCounter(SimStruct *S, (void *)elapseTime)`: This function is provided for the use of blocks that require the elapsed time values for fixed-point computations. `ssGetElapseTimeCounter` returns, to the location pointed to by `elapseTime`, the integer value of the elapsed time counter associated with the S-function. If the counter size is 64 bits, the value is returned as an array of two 32-bit words, with the low-order word stored at the lower address.

  To determine how to access the returned counter value, obtain the data type of the counter by calling `ssGetElapseTimeCounterDtype`, as in the following code:

```
int    *y_dtype;
ssGetElapseTimeCounterDtype(S, y_dtype);

 switch(*y_dtype) {
  case SS_DOUBLE_UINT32:
      {
          uint32_T dataPtr[2];
          ssGetElapseTimeCounter(S, dataPtr);
      }
      break;
  case SS_UINT32:
```

```
      {
         uint32_T dataPtr[1];
         ssGetElapseTimeCounter(S, dataPtr);
      }
       break;
  case SS_UINT16:
     {
          uint16_T dataPtr[1];
          ssGetElapseTimeCounter(S, dataPtr);
     }
      break;
  case SS_UINT8:
     {
         uint8_T dataPtr[1];
         ssGetElapseTimeCounter(S, dataPtr);
    }
     break;
  case SS_DOUBLE:
      {
         real_T dataPtr[1];
         ssGetElapseTimeCounter(S, dataPtr);
     }
      break;
  default:
    ssSetErrorStatus(S, "Invalid data type for elaspe time
        counter");
    break;
}
```

If you want to use the actual elapsed time, issue a call to the `ssGetElapseTime` function to access the elapsed time directly. You do not need to get the counter value and then calculate the elapsed time.

```
double *y_elapseTime;
.
.
.
ssGetElapseTime(S, elapseTime)
```

TLC API for Code Generation

The following TLC functions support elapsed time counters in generated code when you inline S-functions by writing TLC scripts for them.

- `LibGetTaskTimeFromTID(block)`: Generates code to read the absolute time for the task in which `block` executes.

  `LibGetTaskTimeFromTID` is documented with other sample time functions in the TLC Function Library Reference pages of the Target Language Compiler documentation.

  Note Do not use `LibGetT` for this purpose. `LibGetT` always reads the base rate (`tid 0`) timer. If `LibGetT` is called for a block executing at a subrate, the wrong timer is read, causing serious errors.

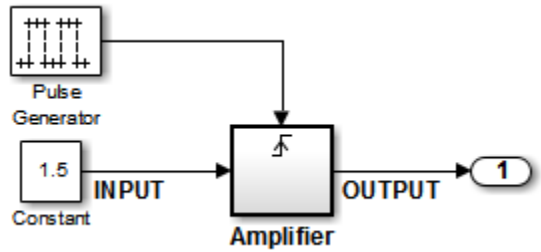
- `LibGetElapseTime(system)`: Generates code to read the elapsed time counter for `system`. (`system` is the parent system of the calling block.) See "Generate Code for an Elapsed Time Counter" on page 15-9 for an example of code generated by this function.
- `LibGetElapseTimeCounter(system)`: Generates code to read the integer value of the elapsed time counter for `system`. (`system` is the parent system of the calling block.) This function should be used in conjunction with `LibGetElapseTimeCounterDtypeId` and `LibGetElapseTimeResolution`. (See the discussion of `ssGetElapseTimeCounter` above.)
- `LibGetElapseTimeCounterDtypeId(system)`: Generates code that returns the data type of the elapsed time counter for `system`. (`system` is the parent system of the calling block.)
- `LibGetElapseTimeResolution(system)`: Generates code that returns the resolution of the elapsed time counter for `system`. (`system` is the parent system of the calling block.)

More About

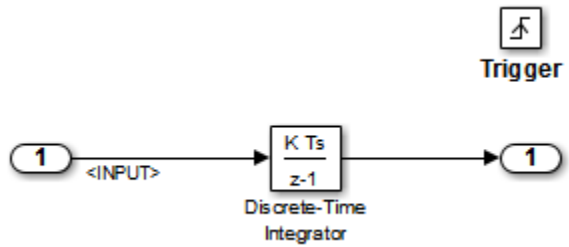
- "Absolute and Elapsed Time Computation" (Simulink Coder)
- "Generate Code for an Elapsed Time Counter" (Simulink Coder)
- "Absolute Time Limitations" (Simulink Coder)

Generate Code for an Elapsed Time Counter

This example shows a model that includes a triggered subsystem, `Amplifier`, consisting of a Discrete-Time Integrator block that uses an elapsed time counter. The model `ex_elapseTime` is in the folder `matlab/help/toolbox/rtw/examples`.



ex_elapseTime Model



Amplifier Subsystem

Code in the generated header file `ex_elapseTime.h` for the model uses 64 bits to implement the timer for the base rate (`clockTick0` and `clockTickH0`).

```
/*
 * Timing:
 * The following substructure contains information regarding
 * the timing information for the model.
 */
struct {
  time_T taskTime0;
  uint32_T clockTick0;
```

```
  uint32_T clockTickH0;
  time_T stepSize0;
  time_T tFinal;
  boolean_T stopRequestedFlag;
} Timing;
```

The code generator allocates storage for the previous-time value and elapsed-time value of the `Amplifier` subsystem (`Amplifier_PREV_T`) in the `D_Work(states)` structure in `ex_elapsedTime.h`.

```
/* Block states (auto storage) for system '<Root>' */
typedef struct {
  real_T DiscreteTimeIntegrator_DSTATE;/* '<S1>/Discrete-Time Integrator' */
  int32_T clockTickCounter;            /* '<Root>/Pulse Generator' */
  uint32_T Amplifier_ELAPS_T[2];       /* '<Root>/Amplifier' */
  uint32_T Amplifier_PREV_T[2];        /* '<Root>/Amplifier' */
 } DW_ex_elapseTime_T;
```

The elapsed time computation is performed as follows within the `ex_elapseTime_step` function:

```
 /* Outputs for Triggered SubSystem: '<Root>/Amplifier' incorporates:
  *  TriggerPort: '<S1>/Trigger'
  */
 zcEvent = rt_ZCFcn(RISING_ZERO_CROSSING,
        &ex_elapseTime_PrevZCX.Amplifier_Trig_ZCE,
        ((real_T)rtb_PulseGenerator));
 if (zcEvent != NO_ZCEVENT) {
   elapseT_H = ex_elapseTime_M->Timing.clockTickH0 -
     ex_elapseTime_DW.Amplifier_PREV_T[1];
   if (ex_elapseTime_DW.Amplifier_PREV_T[0] >
       ex_elapseTime_M->Timing.clockTick0) {
     elapseT_H--;
   }

   ex_elapseTime_DW.Amplifier_ELAPS_T[0] = ex_elapseTime_M->Timing.clockTick0 -
   ex_elapseTime_DW.Amplifier_PREV_T[0];
   ex_elapseTime_DW.Amplifier_PREV_T[0] = ex_elapseTime_M->Timing.clockTick0;
   ex_elapseTime_DW.Amplifier_ELAPS_T[1] = elapseT_H;
   ex_elapseTime_DW.Amplifier_PREV_T[1] = ex_elapseTime_M->Timing.clockTickH0;
```

As shown above, the elapsed time is maintained as a state of the triggered subsystem. The Discrete-Time Integrator block finally performs its output and update computations using the elapsed time.

```
/* DiscreteIntegrator: '<S1>/Discrete-Time Integrator' */
OUTPUT = ex_elapseTime_DW.DiscreteTimeIntegrator_DSTATE;

/* Update for DiscreteIntegrator: '<S1>/Discrete-Time Integrator' incorporates:
 *  Constant: '<Root>/Constant'
 */
ex_elapseTime_DW.DiscreteTimeIntegrator_DSTATE += 0.3 * (real_T)
  ex_elapseTime_DW.Amplifier_ELAPS_T[0] * 1.5;
```

More About

- "Absolute and Elapsed Time Computation" (Simulink Coder)
- "Absolute Time Limitations" (Simulink Coder)

Absolute Time Limitations

Absolute time is the time that has elapsed from the beginning of program execution to the present time, as distinct from *elapsed time*, the interval between two events. See "Absolute and Elapsed Time Computation" on page 15-2 for more information.

When you design an application that is intended to run indefinitely, you must take care when logging time values, or using charts or blocks that depend on absolute time. If the value of time reaches the largest value that can be represented by the data type used by the timer to store time, the timer overflows and the logged time or block output is incorrect.

If your target uses `rtModel`, you can avoid timer overflow by specifying a value for the **Application life span** parameter. See "Integer Timers in Generated Code" on page 15-3 for more information.

The following limitations apply to absolute time:

- If you log time values by opening the Configuration Parameters dialog box and enabling **Data Import/Export > Time** parameter, your model uses absolute time.
- Every Stateflow chart that uses time is dependent on absolute time. The only way to eliminate the dependency is to change the Stateflow chart to not use time.
- The following Simulink blocks depend on absolute time:
  - Backlash
  - Chirp Signal
  - Clock
  - Derivative
  - Digital Clock
  - Discrete-Time Integrator (only when used in triggered subsystems)
  - From File
  - From Workspace
  - Pulse Generator
  - Ramp
  - Rate Limiter
  - Repeating Sequence

- Signal Generator
- Sine Wave (only when the **Sine type** parameter is set to `Time-based`)
- Step
- To File
- To Workspace (only when logging to `StructureWithTime` format)
- Transport Delay
- Variable Time Delay
- Variable Transport Delay

In addition to the Simulink blocks above, blocks in other blocksets may depend on absolute time. See the documentation for the blocksets that you use.

More About

- "Absolute and Elapsed Time Computation" (Simulink Coder)

16

Time-Based Scheduling in Simulink Coder

Time-Based Scheduling and Code Generation

In this section...

Sample Time Considerations

Simulink models run at one or more sample times. The Simulink product provides considerable flexibility in building multirate systems, that is, systems with more than one sample time. However, this same flexibility also allows you to construct models for which the code generator cannot generate real-time code for execution in a multitasking environment. To make multirate models operate as expected in real time (that is, to give the right answers), you sometimes must modify your model or instruct the Simulink engine to modify the model for you. In general, the modifications involve placing Rate Transition blocks between blocks that have unequal sample times. The following sections discuss issues you must address to use a multirate model in a multitasking environment. For a comprehensive discussion of sample times, including rate transitions, see "What Is Sample Time?" (Simulink), "Sample Times in Subsystems" (Simulink), "Sample Times in Systems" (Simulink), "Resolve Rate Transitions" (Simulink), and associated topics.

Tasking Modes

There are two execution modes for a fixed-step Simulink model: single-tasking and multitasking. These modes are available only for fixed-step solvers. To select an execution mode, use the **Treat each discrete rate as a separate task** checkbox on the **Solver** pane of the Configuration Parameters dialog box. When this parameter is selected, multitasking execution is applied for a multirate model. When this option is cleared, single-tasking execution is applied.

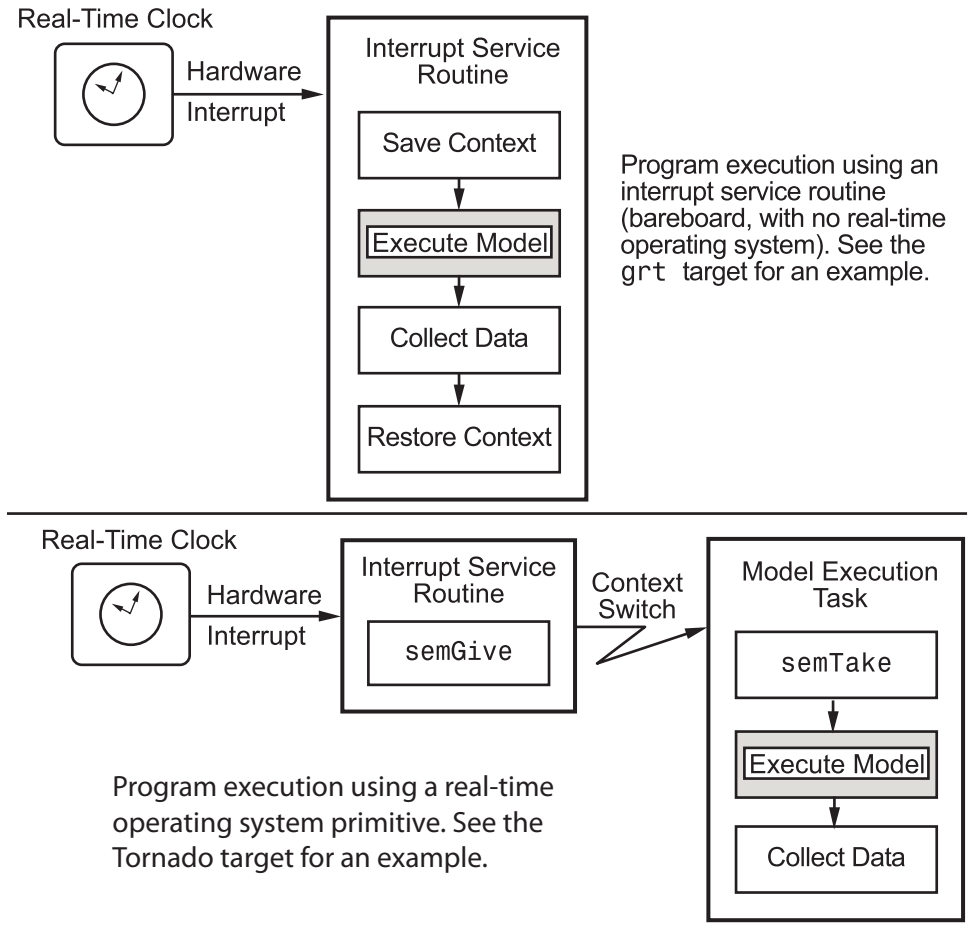
Note: A model that is multirate and uses multitasking cannot reference a multirate model that uses single-tasking.

Execution of models in a real-time system can be done with the aid of a real-time operating system, or it can be done on *bare-metal* target hardware, where the model runs in the context of an interrupt service routine (ISR).

The fact that a system (such as The Open Group UNIX or Microsoft Windows systems) is multitasking does not imply that your program can execute in real time. This is because the program might not preempt other processes when required.

In operating systems (such as PC-DOS) where only one process can exist at a given time, an interrupt service routine (ISR) must perform the steps of saving the processor context, executing the model code, collecting data, and restoring the processor context.

Other operating systems, such as POSIX-compliant ones, provide automatic context switching and task scheduling. This simplifies the operations performed by the ISR. In this case, the ISR simply enables the model execution task, which is normally blocked. The next figure illustrates this difference.



Model Execution and Rate Transitions

To generate code that executes as expected in real time, you (or the Simulink engine) might need to identify and handle sample rate transitions within the model. In multitasking mode, by default the Simulink engine flags errors during simulation if the model contains invalid rate transitions, although you can use the **Multitask rate transition** diagnostic to alter this behavior. A similar diagnostic, called **Single task rate transition**, exists for single-tasking mode.

To avoid raising rate transition errors, insert Rate Transition blocks between tasks. You can request that the Simulink engine handle rate transitions automatically by inserting hidden Rate Transition blocks. See “Automatic Rate Transition” on page 16-25 for an explanation of this option.

To understand such problems, first consider how Simulink simulations differ from real-time programs.

Execution During Simulink Model Simulation

Before the Simulink engine simulates a model, it orders the blocks based upon their topological dependencies. This includes expanding virtual subsystems into the individual blocks they contain and flattening the entire model into a single list. Once this step is complete, each block is executed in order.

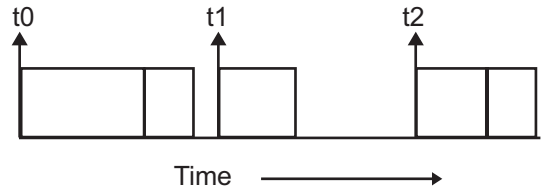
The key to this process is the ordering of blocks. A block whose output is directly dependent on its input (that is, a block with direct feedthrough) cannot execute until the block driving its input executes.

Some blocks set their outputs based on values acquired in a previous time step or from initial conditions specified as a block parameter. The output of such a block is determined by a value stored in memory, which can be updated independently of its input. During simulation, computations are performed prior to advancing the variable corresponding to time. This results in computations occurring instantaneously (that is, no computational delay).

Model Execution in Real Time

A real-time program differs from a Simulink simulation in that the program must execute the model code synchronously with real time. Every calculation results in some computational delay. This means the sample intervals cannot be shortened or lengthened (as they can be in a Simulink simulation), which leads to less efficient execution.

Consider the following timing figure.



Note the processing inefficiency in the sample interval `t1`. That interval cannot be compressed to increase execution speed because, by definition, sample times are clocked in real time.

You can circumvent this potential inefficiency by using the multitasking mode. The multitasking mode defines tasks with different priorities to execute parts of the model code that have different sample rates.

See "Multitasking and Pseudomultitasking Modes" on page 16-12 for a description of how this works. It is important to understand that section before proceeding here.

Single-Tasking Versus Multitasking Operation

Single-tasking programs require longer sample intervals, because all computations must be executed within each clock period. This can result in inefficient use of available CPU time, as shown in the previous figure.

Multitasking mode can improve the efficiency of your program if the model is large and has many blocks executing at each rate.

However, if your model is dominated by a single rate, and only a few blocks execute at a slower rate, multitasking can actually degrade performance. In such a model, the overhead incurred in task switching can be greater than the time required to execute the slower blocks. In this case, it is more efficient to execute all blocks at the dominant rate.

If you have a model that can benefit from multitasking execution, you might need to modify your model by adding Rate Transition blocks (or instruct the Simulink engine to do so) to generate expected results.

For more information about the two modes of execution and examples, see "Modeling for Single-Tasking Execution" on page 16-8 and "Modeling for Multitasking Execution" on page 16-12.

More About

- “What Is Sample Time?” (Simulink)
- “Sample Times in Subsystems” (Simulink)
- “Sample Times in Systems” (Simulink)
- “Configure Time-Based Scheduling” (Simulink Coder)
- “Sample Times in Subsystems” (Simulink)
- “Sample Times in Systems” (Simulink)
- “Resolve Rate Transitions” (Simulink)
- “Handle Rate Transitions” (Simulink Coder)
- “Time-Based Scheduling and Code Generation” (Simulink Coder)
- “Modeling for Single-Tasking Execution” (Simulink Coder)
- “Modeling for Multitasking Execution” (Simulink Coder)
- “Time-Based Scheduling Example Models” (Simulink Coder)

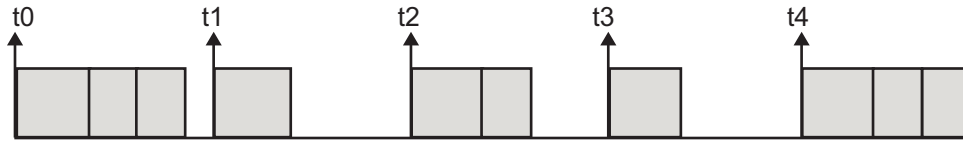
Modeling for Single-Tasking Execution

Single-Tasking Mode

You can execute model code in a strictly single-tasking manner. While this mode is less efficient with regard to execution speed, in certain situations, it can simplify your model.

In single-tasking mode, the base sample rate must define a time interval that is long enough to allow the execution of all blocks within that interval.

The next figure illustrates the inefficiency inherent in single-tasking execution.



Single-tasking system execution requires a base sample rate that is long enough to execute one step through the entire model.

Build a Program for Single-Tasking Execution

To use single-tasking execution, clear the **Treat each discrete rate as a separate task** checkbox on the **Solver** pane of the Configuration Parameters dialog box. If you select the checkbox, single-tasking mode is used in the following cases:

- If your model contains one sample time
- If your model contains a continuous and a discrete sample time and the fixed step size is equal to the discrete sample time

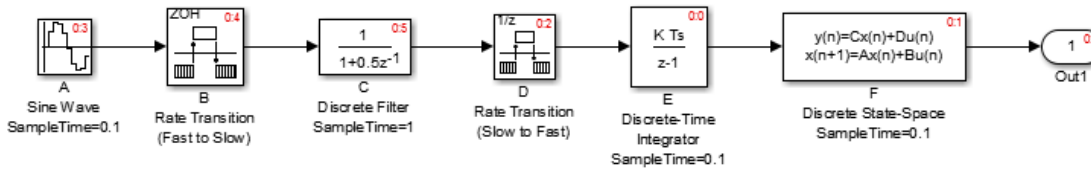
Single-Tasking Execution

This example examines how a simple multirate model executes in both real time and simulation, using a fixed-step solver. It considers operation in both single-tasking and multitasking modes, as determined by setting of the **Treat each discrete rate as a separate task** parameter on the **Solver** pane.

The example model is shown in the next figure. The discussion refers to the six blocks of the model as A through F, as labeled in the block diagram.

The execution order of the blocks (indicated in the upper right of each block) has been forced into the order shown by assigning higher priorities to blocks F, E, and D. The ordering shown is one possible valid execution ordering for this model. For more information, see "Simulation Phases in Dynamic Systems" (Simulink).

The execution order is determined by data dependencies between blocks. In a real-time system, the execution order determines the order in which blocks execute within a given time interval or task. This discussion treats the model's execution order as a given, because it is concerned with the allocation of block computations to tasks, and the scheduling of task execution.



Note The discussion and timing diagrams in this section are based on the assumption that the Rate Transition blocks are used in the default (protected/deterministic) mode, with the **Ensure data integrity during data transfer** and **Ensure deterministic data transfer (maximum delay)** options on.

This example considers the execution of the above model when the **Treat each discrete rate as a separate task** checkbox is cleared, which indicates the single-tasking mode.

In a single-tasking system, if the **Block reduction** option on the **All Parameters** tab is on, fast-to-slow Rate Transition blocks are optimized out of the model. The default case is shown (**Block reduction** on), so block B does not appear in the timing diagrams in this section. For more information, see "Block reduction" (Simulink).

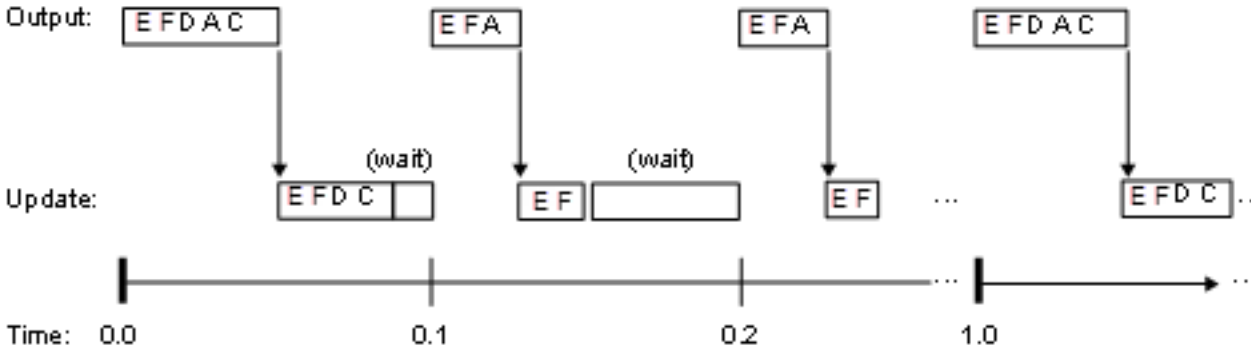
The following table shows, for each block in the model, the execution order, sample time, and whether the block has an output or update computation. Block A does not have discrete states, and accordingly does not have an update computation.

Execution Order and Sample Times (Single-Tasking)

Blocks (in Execution Order)	Sample Time (in Seconds)	Output	Update
E	0.1	Y	Y
F	0.1	Y	Y
D	1	Y	Y
A	0.1	Y	N
C	1	Y	Y

Real-Time Single-Tasking Execution

The next figure shows the scheduling of computations when the generated code is deployed in a real-time system. The generated program is shown running in real time, under control of interrupts from a 10 Hz timer.



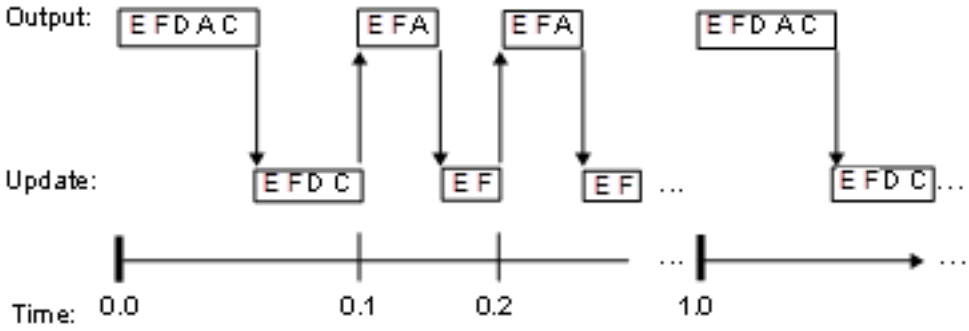
At time 0.0, 1.0, and every second thereafter, both the slow and fast blocks execute their output computations; this is followed by update computations for blocks that have states. Within a given time interval, output and update computations are sequenced in block execution order.

The fast blocks execute on every tick, at intervals of 0.1 second. Output computations are followed by update computations.

The system spends some portion of each time interval (labeled “wait”) idling. During the intervals when only the fast blocks execute, a larger portion of the interval is spent idling. This illustrates an inherent inefficiency of single-tasking mode.

Simulated Single-Tasking Execution

The next figure shows the execution of the model during the Simulink simulation loop.



Because time is simulated, the placement of ticks represents the iterations of the simulation loop. Blocks execute in exactly the same order as in the previous figure, but without the constraint of a real-time clock. Therefore there is no idle time between simulated sample periods.

More About

- "Time-Based Scheduling and Code Generation" (Simulink Coder)
- "Configure Time-Based Scheduling" (Simulink Coder)
- "Time-Based Scheduling Example Models" (Simulink Coder)

Modeling for Multitasking Execution

Multitasking and Pseudomultitasking Modes

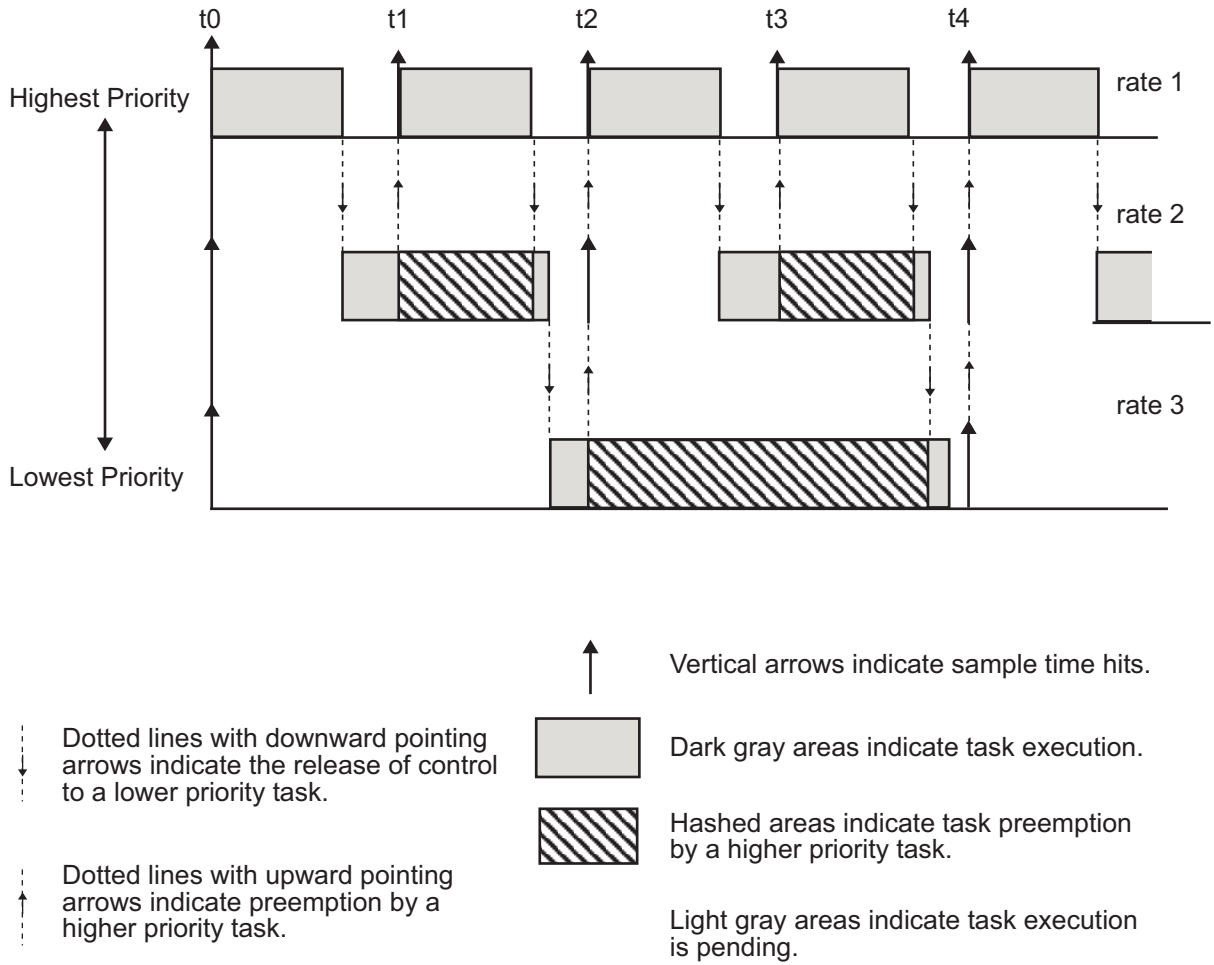
When periodic tasks execute in a multitasking mode, by default the blocks with the fastest sample rates are executed by the task with the highest priority, the next fastest blocks are executed by a task with the next higher priority, and so on. Time available in between the processing of high-priority tasks is used for processing lower priority tasks. This results in efficient program execution.

Where tasks are asynchronous rather than periodic, there may not necessarily be a relationship between sample rates and task priorities; the task with the highest priority need not have the fastest sample rate. You specify asynchronous task priorities using Async Interrupt and Task Sync blocks. You can switch the sense of what priority numbers mean by selecting or deselecting the Solver option **Higher priority value indicates higher task priority**.

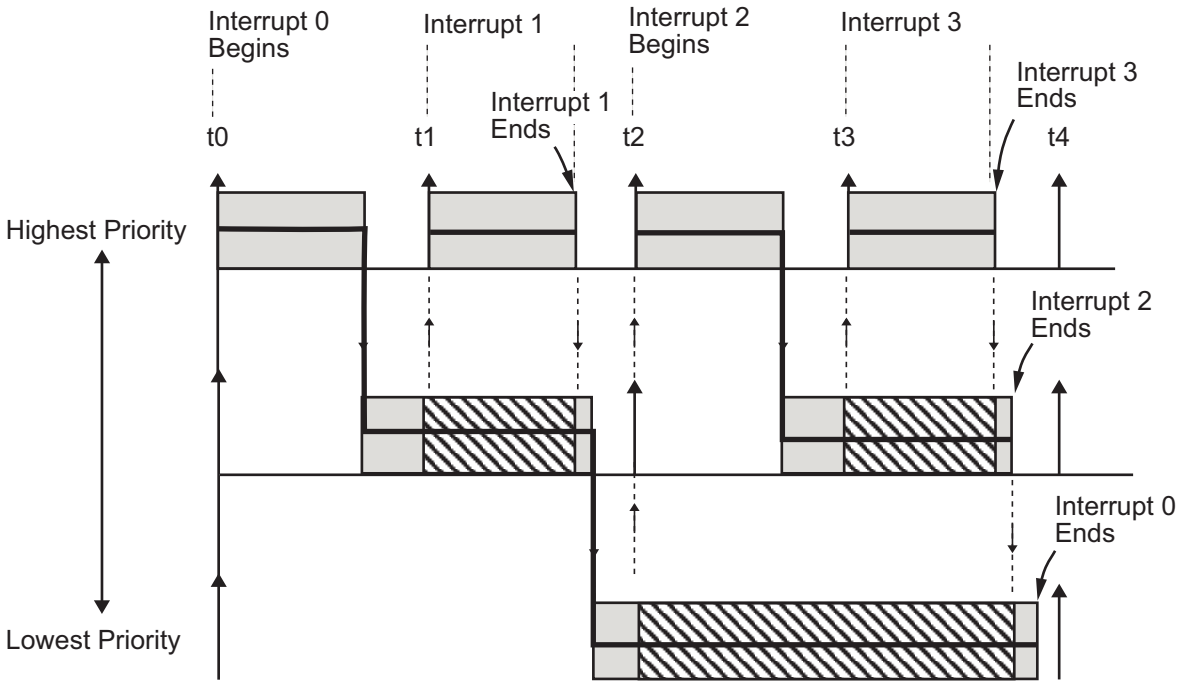
In multitasking environments (that is, under a real-time operating system), you can define separate tasks and assign them priorities. For bare-metal target hardware (that is, no real-time operating system present), you cannot create separate tasks. However, generated application modules implement what is effectively a multitasking execution scheme using overlapped interrupts, accompanied by programmatic context switching.

This means an interrupt can occur while another interrupt is currently in progress. When this happens, the current interrupt is preempted, the floating-point unit (FPU) context is saved, and the higher priority interrupt executes its higher priority (that is, faster sample rate) code. Once complete, control is returned to the preempted ISR.

The next figures illustrate how timing of tasks in multirate systems are handled by the code generator in multitasking, pseudomultitasking, and single-tasking environments.



The next figure shows how overlapped interrupts are used to implement pseudomultitasking. In this case, Interrupt 0 does not return until after Interrupts 1, 2, and 3.



Build a Program for Multitasking Execution

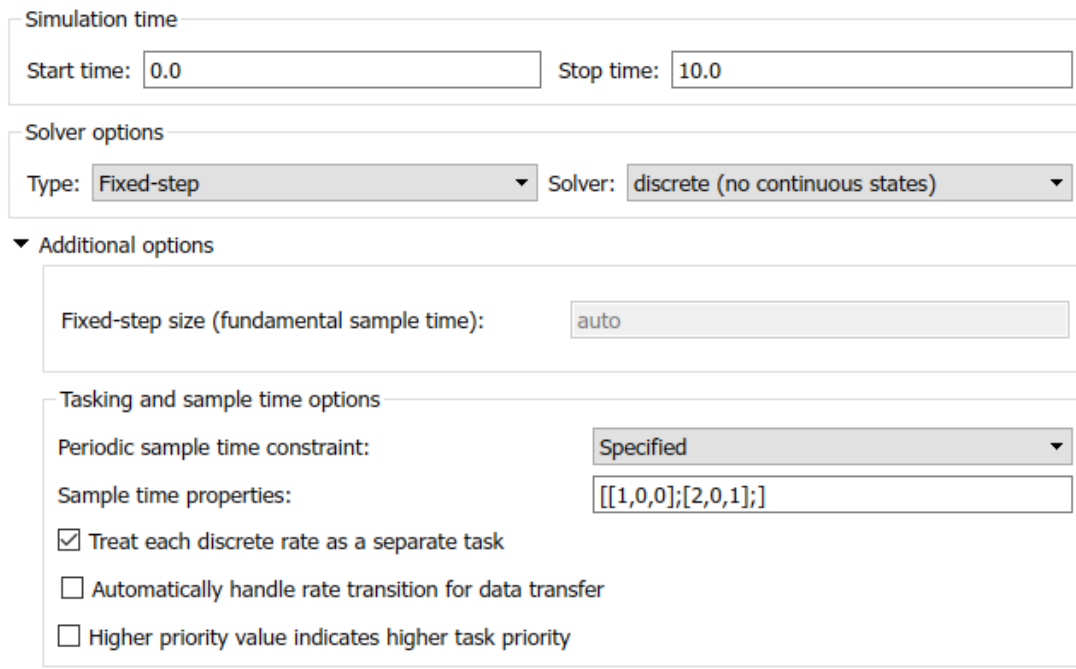
To use multitasking execution, select the **Treat each discrete rate as a separate task** check box on the **Solver** pane of the Configuration Parameters dialog box. This menu is active only if you select `Fixed-step` as the solver type. `Auto` mode results in a multitasking environment if your model has two or more different sample times. A model with a continuous and a discrete sample time runs in single-tasking mode if the fixed-step size is equal to the discrete sample time.

Execute Multitasking Models

In cases where the continuous part of a model executes at a rate that is different from the discrete part, or a model has blocks with different sample rates, the Simulink engine assigns each block a *task identifier* (`tid`) to associate the block with the task that executes at the block's sample rate.

You set sample rates and their constraints on the **Solver** pane of the Configuration Parameters dialog box. To generate code, select `Fixed-step` for the solver type. Certain restrictions apply to the sample rates that you can use:

- The sample rate of a block must be an integer multiple of the base (that is, the fastest) sample period.
- When **Periodic sample time constraint** is `unconstrained`, the base sample period is determined by the **Fixed step size** specified on the **Solvers** pane of the Configuration parameters dialog box.
- When **Periodic sample time constraint** is `Specified`, the base rate fixed-step size is the first element of the sample time matrix that you specify in the companion option **Sample time properties**. The **Solver** pane from the example model `rtwdemo_mrmtbb` shows an example.



- Continuous blocks execute by using an integration algorithm that runs at the base sample rate. The base sample period is the greatest common denominator of all rates in the model only when **Periodic sample time constraint** is set to `Unconstrained` and **Fixed step size** is `Auto`.

- The continuous and discrete parts of the model can execute at different rates only if the discrete part is executed at the same or a slower rate than the continuous part and is an integer multiple of the base sample rate.

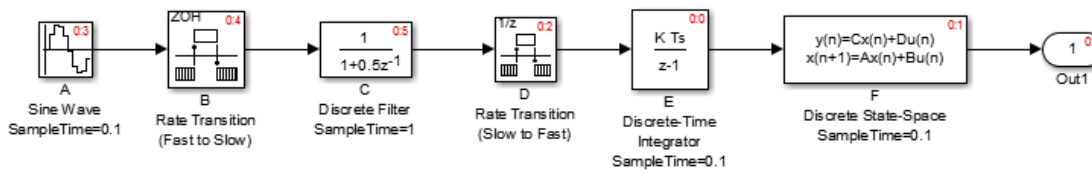
Multitasking Execution

This example examines how a simple multirate model executes in both real time and simulation, using a fixed-step solver. It considers operation in both single-tasking and multitasking modes, as determined by setting of the **Treat each discrete rate as a separate task** parameter on the **Solver** pane.

The example model is shown in the next figure. The discussion refers to the six blocks of the model as A through F, as labeled in the block diagram.

The execution order of the blocks (indicated in the upper right of each block) has been forced into the order shown by assigning higher priorities to blocks F, E, and D. The ordering shown is one possible valid execution ordering for this model. For more information, see "Simulation Phases in Dynamic Systems" (Simulink).

The execution order is determined by data dependencies between blocks. In a real-time system, the execution order determines the order in which blocks execute within a given time interval or task. This discussion treats the model's execution order as a given, because it is concerned with the allocation of block computations to tasks, and the scheduling of task execution.



Note The discussion and timing diagrams in this section are based on the assumption that the Rate Transition blocks are used in the default (protected/deterministic) mode, with the **Ensure data integrity during data transfer** and **Ensure deterministic data transfer (maximum delay)** options on.

This example considers the execution of the above model when the solver **Tasking mode** is `MultiTasking`. Block computations are executed under two tasks, prioritized by rate:

- The slower task, which gets the lower priority, is scheduled to run every second. This is called the *1 second task*.
- The faster task, which gets higher priority, is scheduled to run 10 times per second. This is called the *0.1 second task*. The 0.1 second task can preempt the 1 second task.

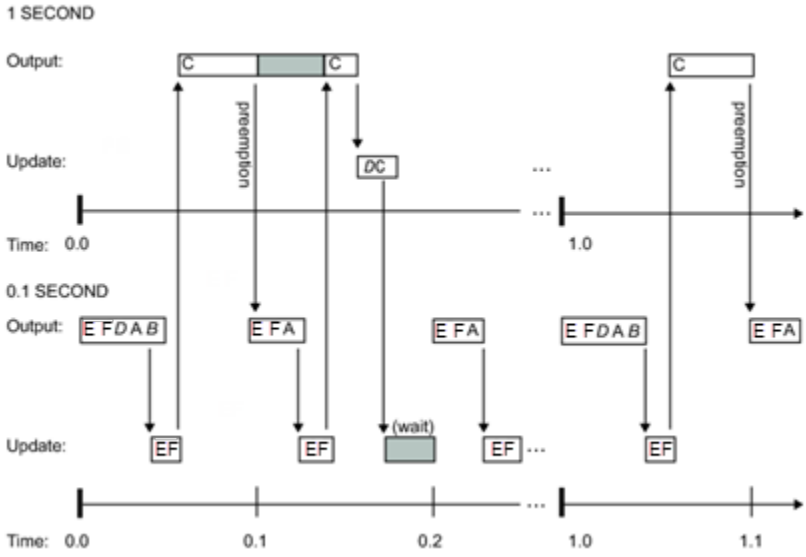
The following table shows, for each block in the model, the execution order, the task under which the block runs, and whether the block has an output or update computation. Blocks A and B do not have discrete states, and accordingly do not have an update computation.

Task Allocation of Blocks in Multitasking Execution

Blocks (in Execution Order)	Task	Output	Update
E	0.1 second task	Y	Y
F	0.1 second task	Y	Y
D	The Rate Transition block uses port-based sample times. Output runs at the output port sample time under 0.1 second task. Update runs at input port sample time under 1 second task. For more information on port-based sample times, see “Sample Times for Model Referencing” (Simulink).	Y	Y
A	0.1 second task	Y	N
B	The Rate Transition block uses port-based sample times. Output runs at the output port sample time under 0.1 second task. For more information on port-based sample times, see “Sample Times for Model Referencing” (Simulink).	Y	N
C	1 second task	Y	Y

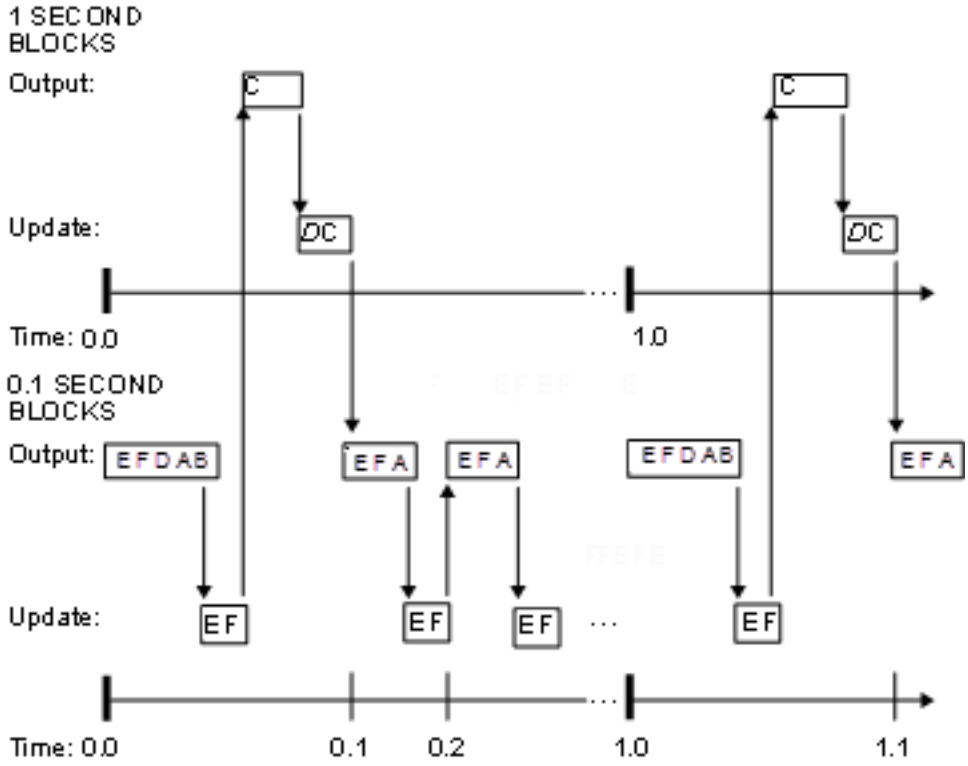
Real-Time Multitasking Execution

The next figure shows the scheduling of computations in `MultiTasking` solver mode when the generated code is deployed in a real-time system. The generated program is shown running in real time, as two tasks under control of interrupts from a 10 Hz timer.



Simulated Multitasking Execution

The next figure shows the Simulink execution of the same model, in `MultiTasking` solver mode. In this case, the Simulink engine runs the blocks in one thread of execution, simulating multitasking. No preemption occurs.



More About

- “Time-Based Scheduling and Code Generation” (Simulink Coder)
- “Sample Times in Subsystems” (Simulink)
- “Sample Times in Systems” (Simulink)
- “Configure Time-Based Scheduling” (Simulink Coder)
- “Resolve Rate Transitions” (Simulink)
- “Handle Rate Transitions” (Simulink Coder)
- “Time-Based Scheduling Example Models” (Simulink Coder)

Handle Rate Transitions

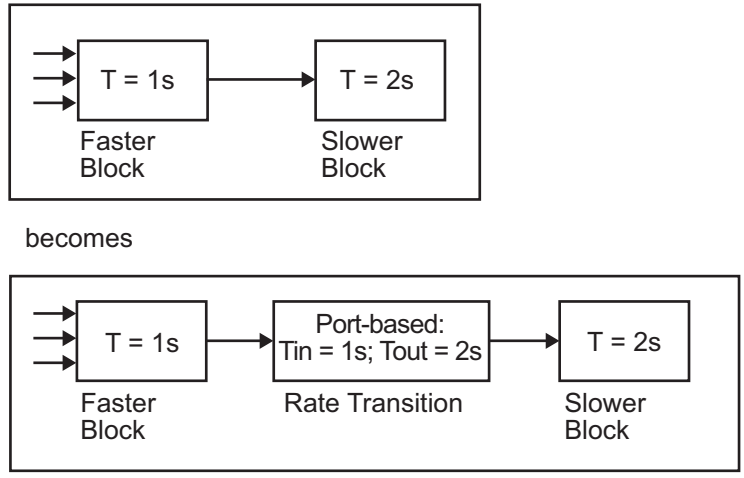
Rate Transitions

Two periodic sample rate transitions can exist within a model:

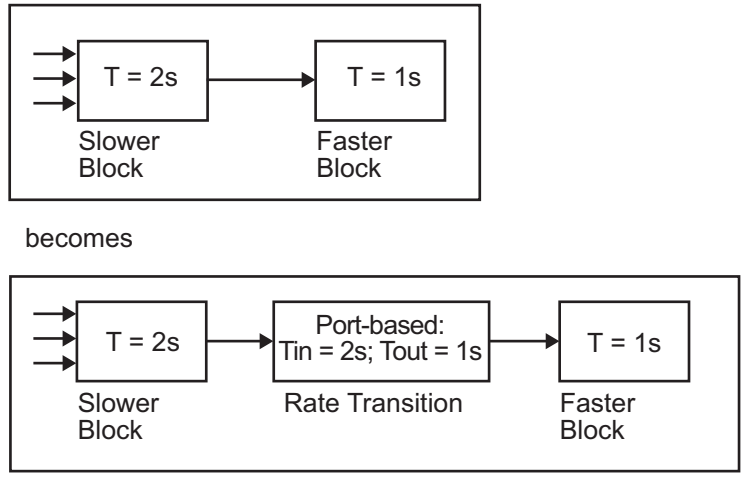
- A faster block driving a slower block
- A slower block driving a faster block

The following sections concern models with periodic sample times with zero offset only. Other considerations apply to multirate models that involve asynchronous tasks. For details on how to generate code for asynchronous multitasking, see "Asynchronous Support" (Simulink Coder).

In multitasking and pseudomultitasking systems, differing sample rates can cause blocks to be executed in the wrong order. To prevent possible errors in calculated data, you must control model execution at these transitions. When connecting faster and slower blocks, you or the Simulink engine must add Rate Transition blocks between them. Fast-to-slow transitions are illustrated in the next figure.



Slow-to-fast transitions are illustrated in the next figure.



Note: Although the Rate Transition block offers a superset of the capabilities of the Unit Delay block (for slow-to-fast transitions) and the Zero-Order Hold block (for fast-to-slow transitions), you should use the Rate Transition block instead of these blocks.

Data Transfer Problems

Rate Transition blocks deal with issues of data integrity and determinism associated with data transfer between blocks running at different rates.

- *Data integrity*: A problem of data integrity exists when the input to a block changes during the execution of that block. Data integrity problems can be caused by preemption.

  Consider the following scenario:

  - A faster block supplies the input to a slower block.
  - The slower block reads an input value V_1 from the faster block and begins computations using that value.
  - The computations are preempted by another execution of the faster block, which computes a new output value V_2.
  - A data integrity problem now arises: when the slower block resumes execution, it continues its computations, now using the “new” input value V_2.

Such a data transfer is called *unprotected*. "Faster to Slower Transitions in Real Time" on page 16-29 shows an unprotected data transfer.

In a *protected* data transfer, the output V_1 of the faster block is held until the slower block finishes executing.

- *Deterministic* versus *nondeterministic* data transfer: In a *deterministic* data transfer, the timing of the data transfer is completely predictable, as determined by the sample rates of the blocks.

  The timing of a *nondeterministic* data transfer depends on the availability of data, the sample rates of the blocks, and the time at which the receiving block begins to execute relative to the driving block.

You can use the Rate Transition block to protect data transfers in your application and make them deterministic. These characteristics are considered desirable in most applications. However, the Rate Transition block supports flexible options that allow you to compromise data integrity and determinism in favor of lower latency. The next section summarizes these options.

Data Transfer Assumptions

When processing data transfers between tasks, the code generator makes these assumptions:

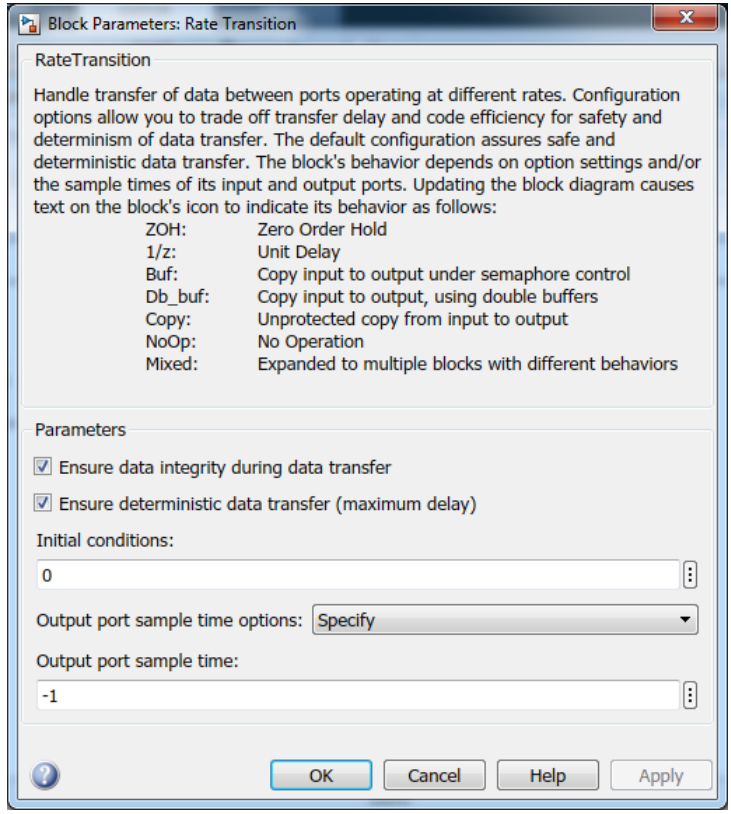
- Data transitions occur between a single reading task and a single writing task.
- A read or write of a byte-sized variable is atomic.
- When two tasks interact through a data transition, only one of them can preempt the other.
- For periodic tasks, the faster rate task has higher priority than the slower rate task; the faster rate task preempts the slower rate task.
- All tasks run on a single processor. Time slicing is not allowed.
- Processes do not crash or restart (especially while data is transferred between tasks).

Rate Transition Block Options

Several parameters of the Rate Transition block are relevant to its use in code generation for real-time execution, as discussed below. For a complete block description, see Rate Transition.

The Rate Transition block handles periodic (fast to slow and slow to fast) and asynchronous transitions. When inserted between two blocks of differing sample rates, the Rate Transition block automatically configures its input and output sample rates for the type of transition; you do not need to specify whether a transition is slow-to-fast or fast-to-slow (low-to-high or high-to-low priorities for asynchronous tasks).

The critical decision you must make in configuring a Rate Transition block is the choice of data transfer mechanism to be used between the two rates. Your choice is dictated by considerations of safety, memory usage, and performance. As the Rate Transition block parameter dialog box in the next figure shows, the data transfer mechanism is controlled by two options.



- **Ensure data integrity during data transfer**: When this option is on, data transferred between rates maintains its integrity (the data transfer is protected).

When this option is off, the data might not maintain its integrity (the data transfer is unprotected). By default, **Ensure data integrity during data transfer** is on.

- **Ensure deterministic data transfer (maximum delay)**: This option is supported for periodic tasks with an offset of zero and fast and slow rates that are multiples of each other. Enable this option for protected data transfers (when **Ensure data integrity during data transfer** is on). When this option is on, the Rate Transition block behaves like a Zero-Order Hold block (for fast to slow transitions) or a Unit Delay block (for slow to fast transitions). The Rate Transition block controls the timing of data transfer in a completely predictable way. When this option is off, the data transfer is nondeterministic. By default, **Ensure deterministic data transfer (maximum delay)** is on for transitions between periodic rates with an offset of zero; for asynchronous transitions, it cannot be selected.

Thus the Rate Transition block offers three modes of operation with respect to data transfer. In order of level of safety:

- **Protected/Deterministic (default):** This is the safest mode. The drawback of this mode is that it introduces deterministic latency into the system for the case of slow-to-fast periodic rate transitions. For that case, the latency introduced by the Rate Transition block is one sample period of the slower task. For the case of fast-to-slow periodic rate transitions, the Rate Transition block introduces no additional latency.
- **Protected/NonDeterministic:** In this mode, for slow-to-fast periodic rate transitions, data integrity is protected by double-buffering data transferred between rates. For fast-to-slow periodic rate transitions, a semaphore flag is used. The blocks downstream from the Rate Transition block use the latest available data from the block that drives the Rate Transition block. Maximum latency is less than or equal to one sample period of the faster task.

  The drawbacks of this mode are its nondeterministic timing. The advantage of this mode is its low latency.
- **Unprotected/NonDeterministic:** This mode is not recommended for mission-critical applications. The latency of this mode is the same as for Protected/NonDeterministic mode, but memory requirements are reduced since neither double-buffering nor semaphores are required. That is, the Rate Transition block does nothing in this mode other than to pass signals through; it simply exists to notify you that a rate transition exists (and can cause generated code to compute incorrect answers). Selecting this mode, however, generates the least amount of code.

Note In unprotected mode (**Ensure data integrity during data transfer** option off), the Rate Transition block does nothing other than allow the rate transition to exist in the model.

Rate Transition Blocks and Continuous Time

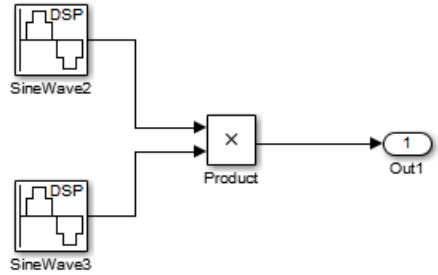
The sample time at the output port of a Rate Transition block can only be discrete or fixed in minor time step. This means that when a Rate Transition block inherits continuous sample time from its destination block, it treats the inherited sample time as Fixed in Minor Time Step. Therefore, the output function of the Rate Transition block runs only at major time steps. If the destination block sample time is continuous, Rate Transition block output sample time is the base rate sample time (if solver is fixed-step), or zero-order-hold-continuous sample time (if solver is variable-step).

Automatic Rate Transition

The Simulink engine can detect mismatched rate transitions in a multitasking model during an update diagram and automatically insert Rate Transition blocks to handle them. To enable this, in the **Solver** pane of model configuration parameters, select **Automatically handle rate transition for data transfer**. The default setting for this option is off. When you select this option:

- Simulink handles transitions between periodic sample times and asynchronous tasks.
- Simulink inserts hidden Rate Transition blocks in the block diagram.
- The code generator produces code for the Rate Transition blocks that were automatically inserted. This code is identical to the code generated for Rate Transition blocks that were inserted manually.
- Automatically inserted Rate Transition blocks operate in protected mode for periodic tasks and asynchronous tasks. You cannot alter this behavior. For periodic tasks, automatically inserted Rate Transition blocks operate with the level of determinism specified by the **Deterministic data transfer** parameter in the **Solver** pane. The default setting is `Whenever possible`, which enables determinism for data transfers between periodic sample-times that are related by an integer multiple. For more information, see “Deterministic data transfer” (Simulink). To use other modes, you must insert Rate Transition blocks and set their modes manually.

For example, in this model, SineWave2 has a sample time of 2, and SineWave3 has a sample time of 3.

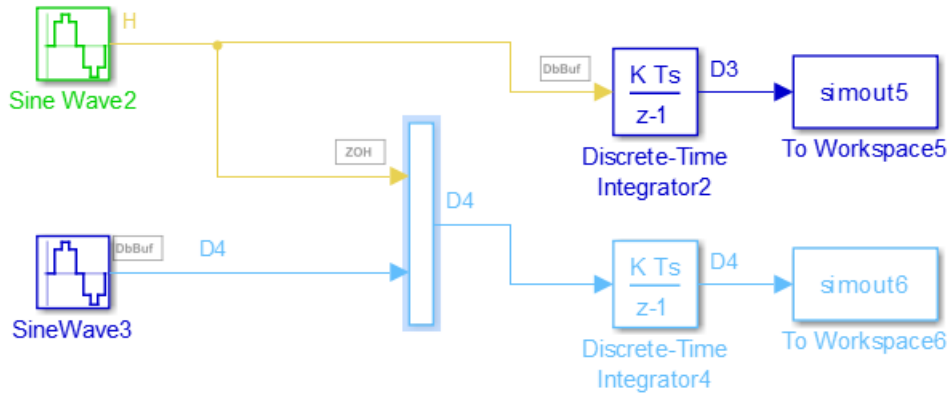


When you select **Automatically handle rate transition for data transfer**, Simulink inserts a Rate Transition block between each Sine Wave block and the Product block. The inserted blocks have the parameter values to reconcile the Sine Wave block sample times.

If the input port and output port data sample rates in a model are not multiples of each other, Simulink inserts a Rate Transition block whose sample rate is the greatest common divisor (GCD) of the two rates. If no other block in the model contains this new rate, an error occurs during simulation. In this case, you must insert a Rate Transition block manually.

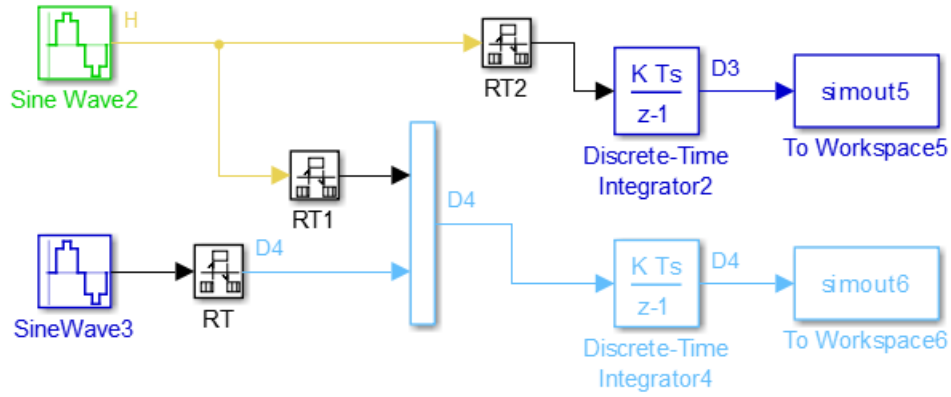
Visualize Inserted Rate Transition Blocks

When you select the **Automatically handle rate transition for data transfer** option, Simulink inserts Rate Transition blocks in the paths that have mismatched transition rates. These blocks are hidden by default. To visualize the inserted blocks, update the diagram. Badge labels appear in the model and indicate where Simulink inserted Rate Transition blocks during the compilation phase. For example, in this model, three Rate Transition blocks were inserted between the two Sine Wave blocks and the Multiplexer and Integrator when the model compiled. The ZOH and DbBuf badge labels indicate these blocks.



You can show or hide badge labels using the **Display > Signals and Ports > Hidden Rate Transition Block Indicators** setting.

To configure the hidden Rate Transition blocks, right click on a badge label and click on **Insert rate transition block** to make the block visible.

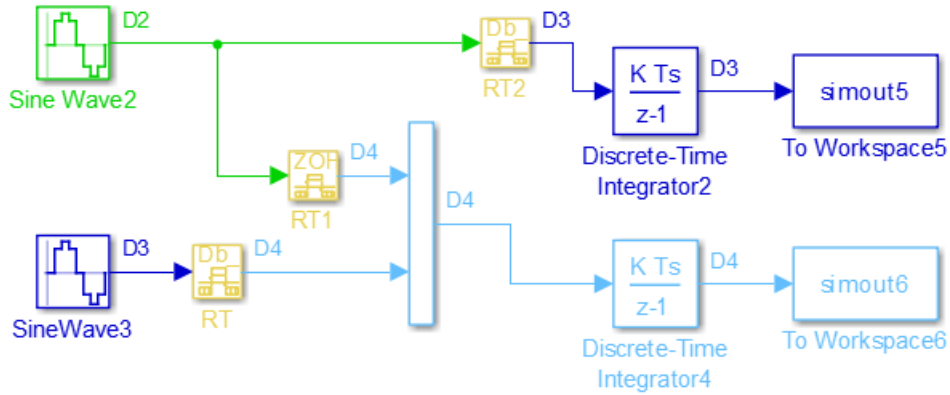


When you make hidden Rate Transition blocks visible:

- You can see the type of Rate Transition block inserted as well as the location in the model.
- You can set the **Initial Conditions** of these blocks.

- You can change block parameters for rate transfer.

Validate the changes to your model by updating your diagram.



Displaying inserted Rate Transition blocks is not compatible with:

- Concurrent execution environment
- Export-function models

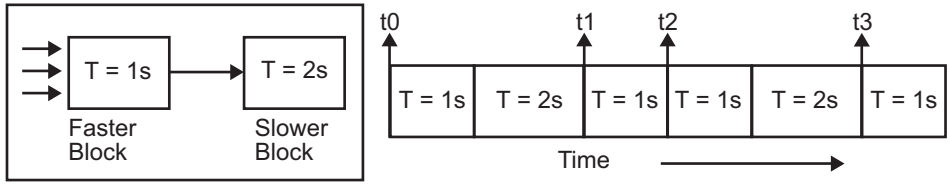
To learn more about the types of Rate Transition blocks, see Rate Transition.

Periodic Sample Rate Transitions

These sections describe cases in which Rate Transition blocks are required for periodic sample rate transitions. The discussion and timing diagrams in these sections are based on the assumption that the Rate Transition block is used in its default (protected/deterministic) mode; that is, the **Ensure data integrity during data transfer** and **Ensure deterministic data transfer (maximum delay)** options are both on. These are the settings used for automatically inserted Rate Transition blocks.

Faster to Slower Transitions in a Simulink Model

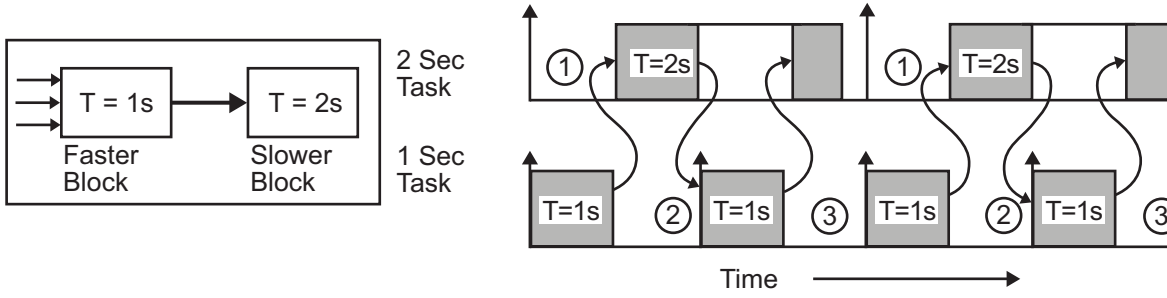
In a model where a faster block drives a slower block having direct feedthrough, the outputs of the faster block are computed first. In simulation intervals where the slower block does not execute, the simulation progresses more rapidly because there are fewer blocks to execute. The next figure illustrates this situation.



A Simulink simulation does not execute in real time, which means that it is not bound by real-time constraints. The simulation waits for, or moves ahead to, whatever tasks are required to complete simulation flow. The actual time interval between sample time steps can vary.

Faster to Slower Transitions in Real Time

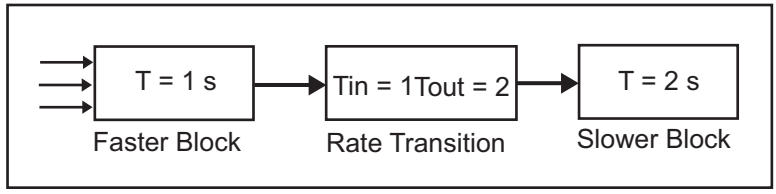
In models where a faster block drives a slower block, you must compensate for the fact that execution of the slower block might span more than one execution period of the faster block. This means that the outputs of the faster block can change before the slower block has finished computing its outputs. The next figure shows a situation in which this problem arises (T = sample time). Note that lower priority tasks are preempted by higher priority tasks before completion.



1. The faster task (T=1s) completes.
2. Higher priority preemption occurs.
3. The slower task (T=2s) resumes and its inputs have changed. This leads to unpredictable results.

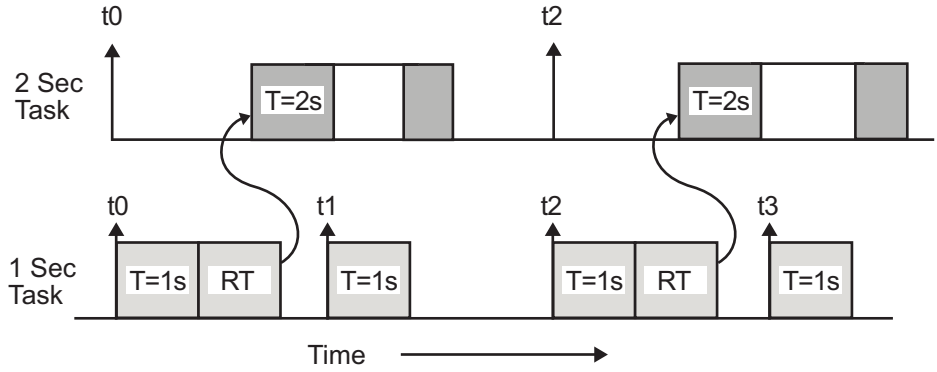
In the above figure, the faster block executes a second time before the slower block has completed execution. This can cause unpredictable results because the input data to the slow task is changing. Data might not maintain its integrity in this situation.

To avoid this situation, the Simulink engine must hold the outputs of the 1 second (faster) block until the 2 second (slower) block finishes executing. The way to accomplish this is by inserting a Rate Transition block between the 1 second and 2 second blocks. The input to the slower block does not change during its execution, maintaining data integrity.



It is assumed that the Rate Transition block is used in its default (protected/ deterministic) mode.

The Rate Transition block executes at the sample rate of the slower block, but with the priority of the faster block.

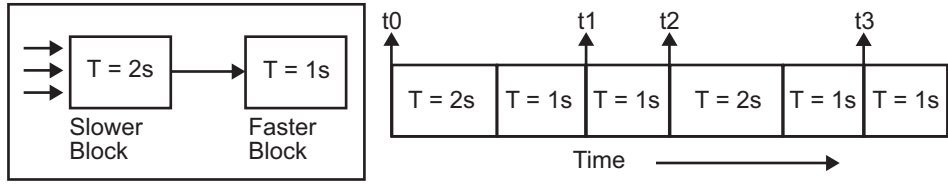


When you add a Rate Transition block, the block executes before the 2 second block (its priority is higher) and its output value is held constant while the 2 second block executes (it executes at the slower sample rate).

Slower to Faster Transitions in a Simulink Model

In a model where a slower block drives a faster block, the Simulink engine again computes the output of the driving block first. During sample intervals where only the faster block executes, the simulation progresses more rapidly.

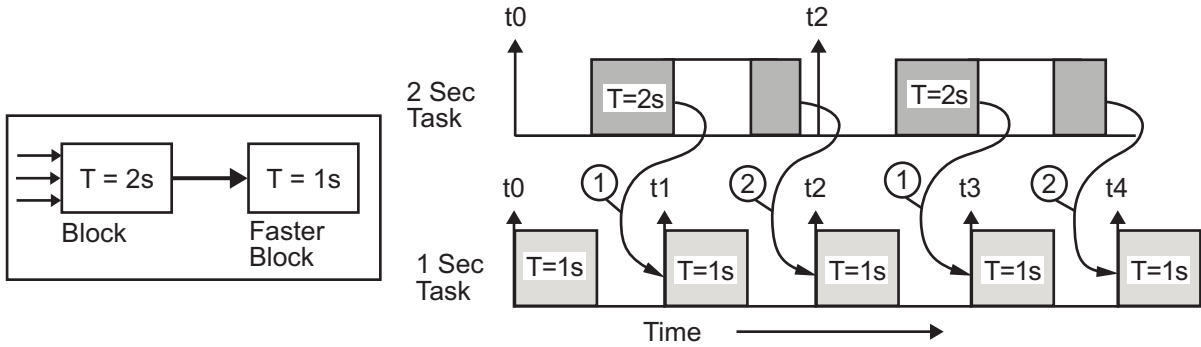
The next figure shows the execution sequence.



As you can see from the preceding figures, the Simulink engine can simulate models with multiple sample rates in an efficient manner. However, a Simulink simulation does not operate in real time.

Slower to Faster Transitions in Real Time

In models where a slower block drives a faster block, the generated code assigns the faster block a higher priority than the slower block. This means the faster block is executed before the slower block, which requires special care to avoid incorrect results.



1 The faster block executes a second time prior to the completion of the slower block.

2 The faster block executes before the slower block.

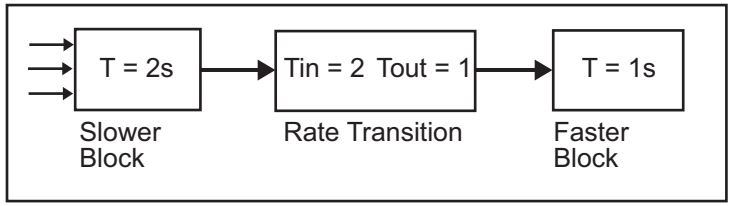
This timing diagram illustrates two problems:

- Execution of the slower block is split over more than one faster block interval. In this case the faster task executes a second time before the slower task has completed

execution. This means the inputs to the faster task can have incorrect values some of the time.

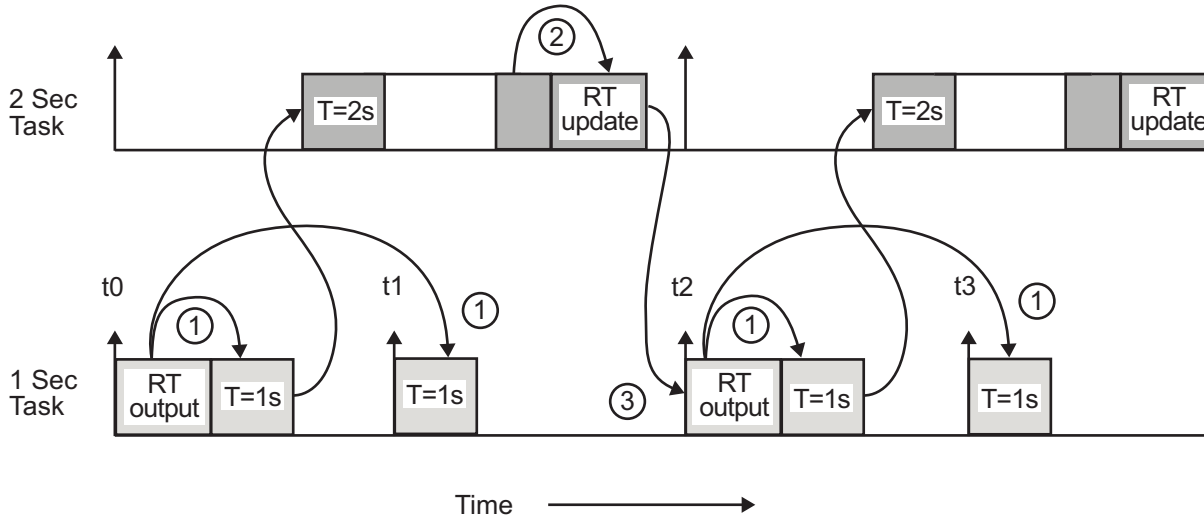
- The faster block executes before the slower block (which is backward from the way a Simulink simulation operates). In this case, the 1 second block executes first; but the inputs to the faster task have not been computed. This can cause unpredictable results.

To eliminate these problems, you must insert a Rate Transition block between the slower and faster blocks.



It is assumed that the Rate Transition block is used in its default (protected/deterministic) mode.

The next figure shows the timing sequence that results with the added Rate Transition block.



Three key points about transitions in this diagram (refer to circled numbers):

1. The Rate Transition block output runs in the 1 second task, but at a slower rate (2 seconds). The output of the Rate Transition block feeds the 1 second task blocks.
2. The Rate Transition update uses the output of the 2 second task to update its internal state.
3. The Rate Transition output in the 1 second task uses the state of the Rate Transition that was updated in the 2 second task.

The first problem is alleviated because the Rate Transition block is updating at a slower rate and at the priority of the slower block. The input to the Rate Transition block (which is the output of the slower block) is read after the slower block completes executing.

The second problem is alleviated because the Rate Transition block executes at a slower rate and its output does not change during the computation of the faster block it is driving. The output portion of a Rate Transition block is executed at the sample rate of the slower block, but with the priority of the faster block. Since the Rate Transition block drives the faster block and has effectively the same priority, it is executed before the faster block.

Note This use of the Rate Transition block changes the model. The output of the slower block is now delayed by one time step compared to the output without a Rate Transition block.

More About

- "Time-Based Scheduling and Code Generation" (Simulink Coder)
- "Sample Times in Subsystems" (Simulink)
- "Sample Times in Systems" (Simulink)
- "Modeling for Multitasking Execution" (Simulink Coder)
- "Configure Time-Based Scheduling" (Simulink Coder)
- "Resolve Rate Transitions" (Simulink)
- "Time-Based Scheduling Example Models" (Simulink Coder)

Configure Time-Based Scheduling

For details about solver options, see "Solver Pane" (Simulink).

Configure Start and Stop Times

The **Stop time** (Simulink) must be greater than or equal to the **Start time** (Simulink). If the stop time is zero, or if the total simulation time (`Stop` minus `Start`) is less than zero, the generated program runs for one step. If the stop time is set to `inf`, the generated program runs indefinitely.

When using the GRT or ERT targets, you can override the stop time when running a generated program from the Microsoft Windows command prompt or UNIX[3] command line. To override the stop time that was set during code generation, use the `-tf` switch.

```
model -tf n
```

The program runs for `n` seconds. If `n = inf`, the program runs indefinitely.

Certain blocks have a dependency on absolute time. If you are designing a program that is intended to run indefinitely (**Stop time** = `inf`), and your generated code does not use the `rtModel` data structure (that is, it uses `simstructs` instead), you must not use these blocks. See "Absolute Time Limitations" on page 15-12 for a list of blocks that can potentially overflow timers.

If you know how long an application that depends on absolute time needs to run, you can prevent the timers from overflowing and force the use of optimal word sizes by specifying the **Application lifespan (days)** (Simulink) parameter on the **Optimization** pane. See "Control Memory Allocation for Time Counters" on page 53-11 for details.

Configure the Solver Type

For code generation, you must configure a model to use a fixed-step solver for all targets except the S-function and RSim targets. You can configure the S-function and RSim targets with a fixed-step or variable-step solver.

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

Configure the Tasking Mode

The code generator supports both single-tasking and multitasking modes for periodic sample times. See “Time-Based Scheduling and Code Generation” on page 16-2 for details.

More About

- “Time-Based Scheduling and Code Generation” (Simulink Coder)
- “Sample Times in Subsystems” (Simulink)
- “Sample Times in Systems” (Simulink)
- “Time-Based Scheduling Example Models” (Simulink Coder)

Time-Based Scheduling Example Models

Optimize Memory Usage for Time Counters

This example shows how to optimize the amount of memory that the code generator allocates for time counters. The example optimizes the memory that stores elapsed time, the interval of time between two events.

The code generator represents time counters as unsigned integers. The word size of time counters is based on the setting of the model configuration parameter **Application lifespan (days)**, which specifies the expected maximum duration of time the application runs. You can use this parameter to prevent time counter overflows. The default size is 64 bits.

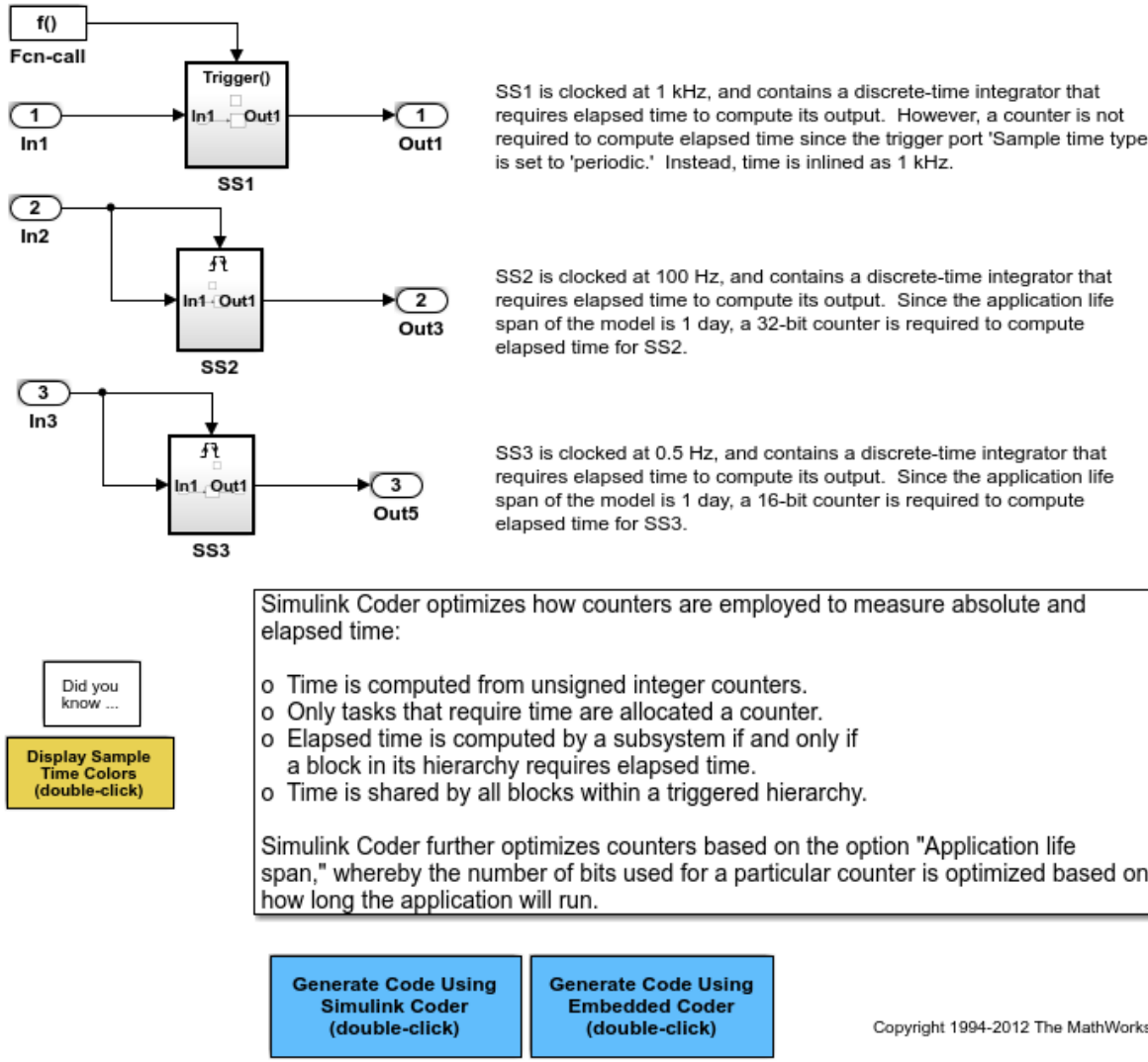
The number of bits that a time counter uses depends on the setting of the **Application lifespan (days)** parameter. For example, if a time counter increments at a rate of 1 kHz, to avoid an overflow, the counter has the following number of bits:

- Lifespan < 0.25 sec: 8 bits
- Lifespan < 1 min: 16 bits
- Lifespan < 49 days: 32 bits
- Lifespan > 50 days: 64 bits

A 64-bit time counter does not overflow for 590 million years.

Open Example Model

Open the example model `rtwdemo_abstime`.



The model consists of three subsystems `SS1`, `SS2`, and `SS3`. On the **Optimization** tab, the **Application lifespan (days)** parameter is set to the default, which is `auto`.

The three subsystems contain a discrete-time integrator that requires elapsed time as input to compute its output value. The subsystems vary as follows:

- SS1 - Clocked at 1 kHz. Does not require a time counter. **Sample time type** parameter for trigger port is set to `periodic`. Elapsed time is inlined as 0.001.
- SS2 - Clocked at 100 Hz. Requires a time counter. Based on a lifespan of 1 day, a 32-bit counter stores the elapsed time.
- SS3 - Clocked at 0.5 Hz. Requires a time counter. Based on a lifespan of 1 day, a 16-bit counter stores the elapsed time.

Simulate the Model

Simulate the model. By default, the model is configured to show sample times in different colors. Discrete sample times for the three subsystems appear red, green, and blue. Triggered subsystems are blue-green.

Generate Code and Report

1. Create a temporary folder for the build and inspection process.

2. Configure the model for the code generator to use the GRT system target file and a lifespan of `inf` days.

3. Build the model.

```
### Starting build procedure for model: rtwdemo_abstime
### Successful completion of build procedure for model: rtwdemo_abstime
```

Review Generated Code

Open the generated source file `rtwdemo_abstime.c`.

```
struct tag_RTM_rtwdemo_abstime_T {
  const char_T *errorStatus;

  /*
   * Timing:
   * The following substructure contains information regarding
   * the timing information for the model.
   */
  struct {
    uint32_T clockTick1;
    uint32_T clockTickH1;
    uint32_T clockTick2;
    uint32_T clockTickH2;
```

```
    struct {
      uint16_T TID[3];
      uint16_T cLimit[3];
    } TaskCounters;
  } Timing;
};
```

Four 32-bit unsigned integers, `clockTick1` , `clockTickH1` , `clockTick2` , and `clockTickH2` are counters for storing the elapsed time of subsystems `SS2` and `SS3`.

Enable Optimization and Regenerate Code

1. Reconfigure the model to set the lifespan to 1 day.

2. Build the model.

```
### Starting build procedure for model: rtwdemo_abstime
### Successful completion of build procedure for model: rtwdemo_abstime
```

Review the Regenerated Code

```
struct tag_RTM_rtwdemo_abstime_T {
  const char_T *errorStatus;

  /*
   * Timing:
   * The following substructure contains information regarding
   * the timing information for the model.
   */
  struct {
    uint32_T clockTick1;
    uint16_T clockTick2;
    struct {
      uint16_T TID[3];
      uint16_T cLimit[3];
    } TaskCounters;
  } Timing;
};
```

The new setting for the **Application lifespan (days)** parameter instructs the code generator to set aside less memory for the time counters. The regenerated code includes:

- 32-bit unsigned integer, `clockTick1`, for storing the elapsed time of the task for `SS2`
- 16-bit unsigned integer, `clockTick2`, for storing the elapsed time of the task for `SS3`

Related Information

- "Optimization Pane: General" (Simulink)
- "Timers in Asynchronous Tasks" (Simulink Coder)
- "Time-Based Scheduling and Code Generation" (Simulink Coder)

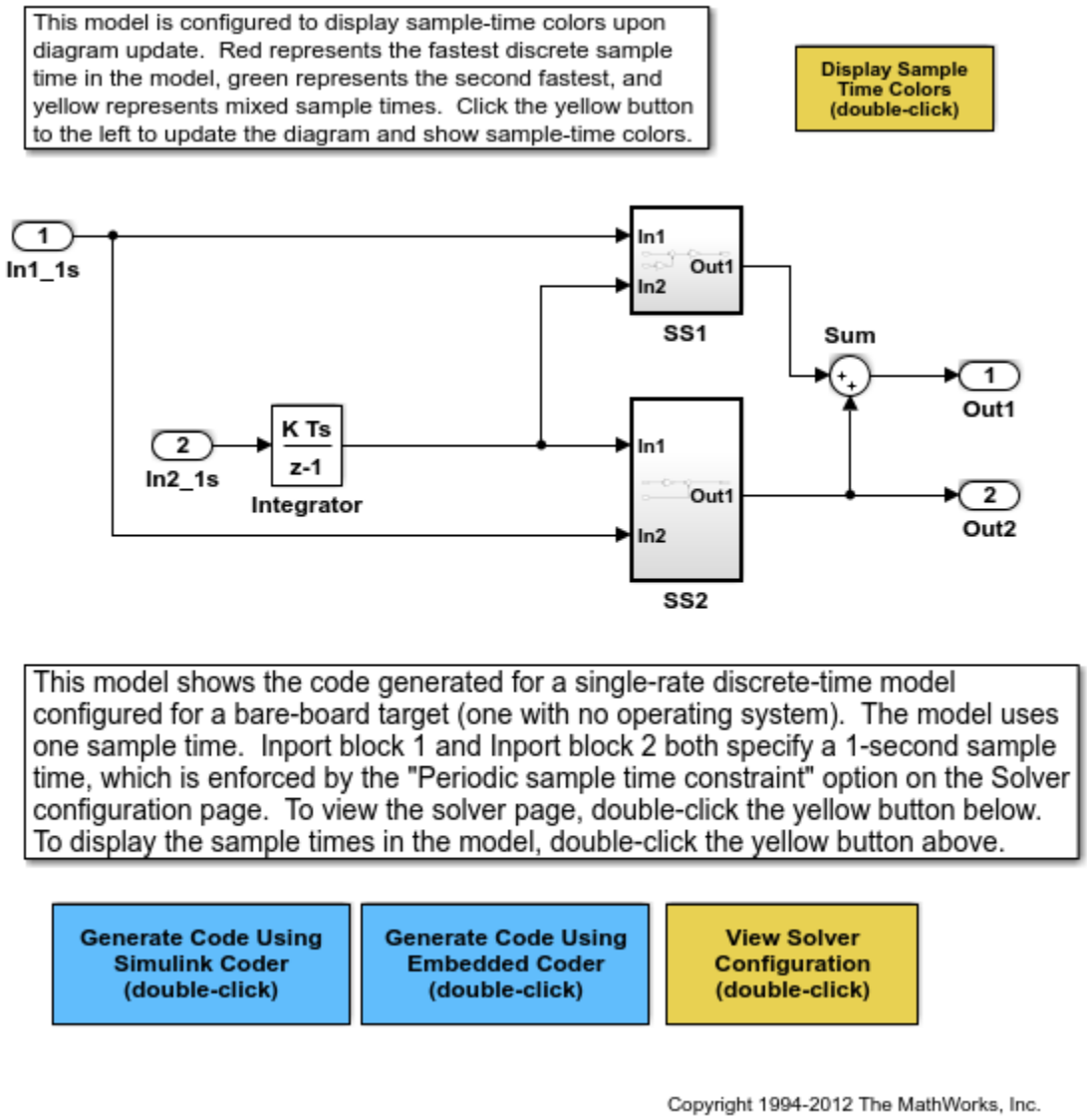
Single-Rate Modeling (Bare Board, No OS)

This model shows the code generated for a single-rate discrete-time model configured for a bare-board target (one with no operating system).

Open Example Model

Open the example model `rtwdemo_srbb`.

```
open_system('rtwdemo_srbb')
```



The model uses one sample time. Inport block 1 and Inport block 2 both specify a 1-second sample time, which is enforced by the **Periodic sample time constraint** option

on the **Solver** configuration page. To view the solver page, double-click the corresponding yellow button in the model. To display the sample times in the model, double-click the corresponding yellow button in the model.

This model is configured to display sample-time colors upon diagram update. Red represents the fastest discrete sample time in the model, green represents the second fastest, and yellow represents mixed sample times. Click the yellow button in the model to update the diagram and show sample-time colors.

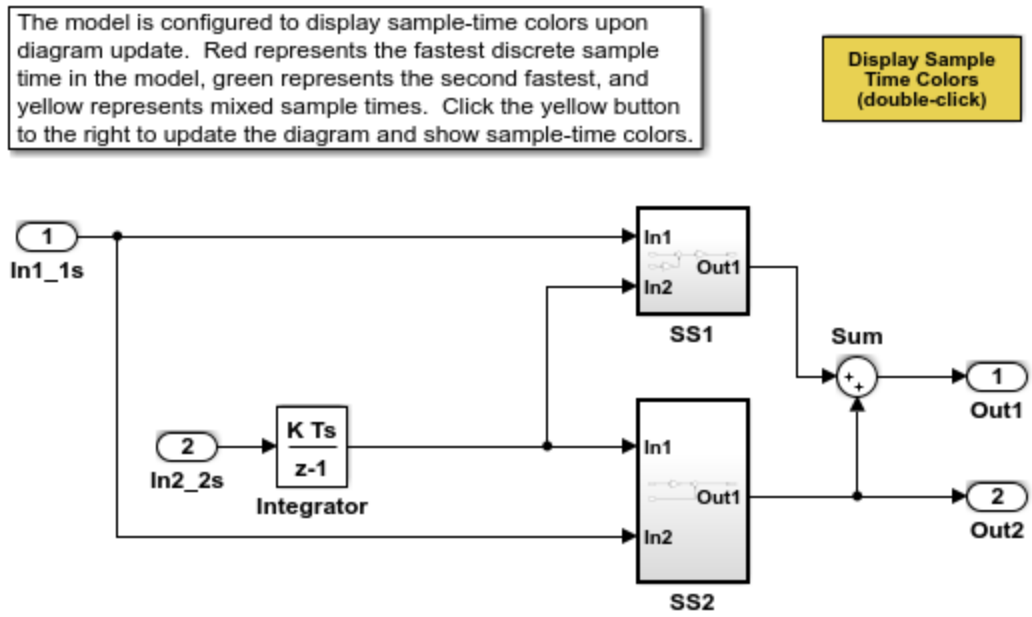
Multirate Modeling in Single-Tasking Mode (Bare Board, no OS)

This model shows the code generated for a multirate discrete-time model configured for single-tasking on a bare-board target (one with no operating system).

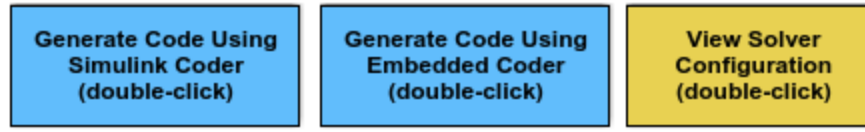
Open Example Model

Open the example model `rtwdemo_mrstbb`.

```
open_system('rtwdemo_mrstbb')
```



This model shows the code generated for a multirate discrete-time model configured for single-tasking on a bare-board target (one with no operating system). The model contains two sample times. Inport block 1 and Inport block 2 specify 1-second and 2-second sample times, respectively, which are enforced by the "Periodic sample time constraint" option on the Solver configuration page. The solver is set for single-tasking operation. Rate transition blocks are, therefore, not necessary between blocks executing at different sample times because preemption will not occur.



The model contains two sample times. Inport block 1 and Inport block 2 specify 1-second and 2-second sample times, respectively, which are enforced by the **Periodic sample time constraint** option on the **Solver** configuration page. The solver is set for single-tasking operation. Rate transition blocks are, therefore, not necessary between blocks executing at different sample times because preemption will not occur.

The model is configured to display sample-time colors upon diagram update. Red represents the fastest discrete sample time in the model, green represents the second fastest, and yellow represents mixed sample times. Double-click the yellow button in the model to update the diagram and show sample-time colors.

Multirate Modeling in Multitasking Mode (Bare Board, no OS)

This model shows the code generated for a multirate discrete-time model configured for a multitasking bare-board target (one with no operating system).

Open Example Model

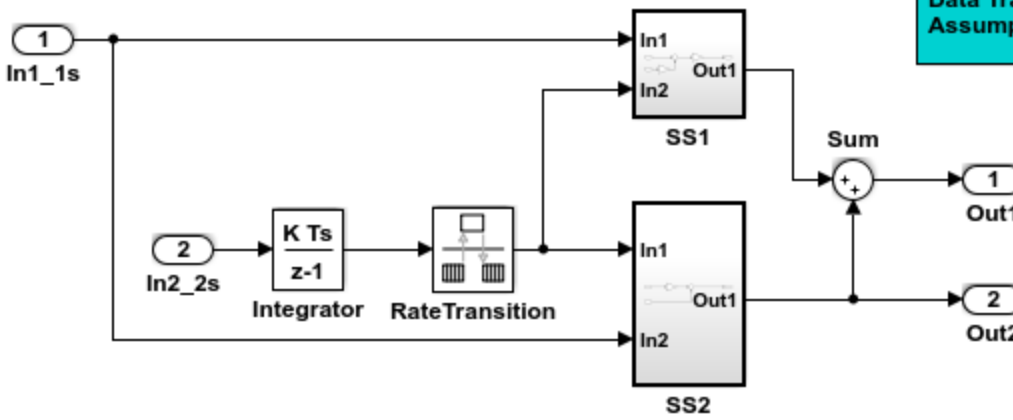
Open the example model `rtwdemo_mrmtbb`.

```
open_system('rtwdemo_mrmtbb')
```

The model is configured to display sample-time colors upon diagram update. Red represents the fastest discrete sample time in the model, green represents the second fastest, and yellow represents mixed sample times. Click the yellow button to the right to update the diagram and show sample-time colors.

Display Sample Time Colors (double-click)

Data Transfer Assumptions ...



This model shows the code generated for a multirate discrete-time model configured for a multitasking bare-board target (one with no operating system). The model contains two sample times. Inport block 1 and Inport block 2 specify 1-second and 2-second sample times, respectively, which are enforced by the "Periodic sample time constraint" option on the Solver configuration page. The solver is set for multitasking operation, which means a rate transition block is required to ensure that data integrity is enforced when the 1-second task preempts the 2-second task. Proper rate transitions are always enforced by Simulink and Simulink Coder. This model specifies an explicit rate transition block. Alternatively, this block could be automatically inserted by Simulink using the "Automatically handle data transfers between tasks" option on the Solver configuration page.

Generate Code Using Simulink Coder (double-click)

Generate Code Using Embedded Coder (double-click)

View Solver Configuration (double-click)

Explore Example Model

The model contains two sample times. Inport block 1 and Inport block 2 specify 1-second and 2-second sample times, respectively, which are enforced by the **Periodic sample time constraint** option on the **Solver** configuration page. The solver is set for multitasking operation, which means a rate transition block is required to ensure that data integrity is enforced when the 1-second task preempts the 2-second task. Proper rate transitions are always enforced by Simulink and Simulink Coder. This model specifies an explicit rate transition block. Alternatively, this block could be automatically inserted by Simulink using the **Automatically handle data transfers between tasks** option on the **Solver** configuration page.

The model is configured to display sample-time colors upon diagram update. Red represents the fastest discrete sample time in the model, green represents the second fastest, and yellow represents mixed sample times. Click the yellow button to the right to update the diagram and show sample-time colors.

Data Transfer Assumptions

Basis of operation for data transfers between tasks:

1. Data transitions occur between a single reading task and a single writing task.
2. A read or write of a byte sized variable is atomic.
3. When two tasks interact through a data transition, only one of them can preempt the other.
4. For periodic tasks, the faster rate task has higher priority than the slower rate task; the faster rate task always preempts the slower rate task.
5. All tasks run on a single processor. Time slicing is not allowed.
6. Processes do not crash/restart (especially while data is being transferred between tasks)

Trade Determinism and Data Integrity to Improve System Performance

This model shows the differences in the operation modes of the Rate Transition block when used in a multirate, multitasking model. The flexible options for the Rate Transition block allow you to select the mode that is best suited for your application. You can trade levels of determinism and data integrity to improve system performance.

Rate Transition Block Modes of Operation

Ensure data integrity and determinism (DetAndInteg) : Data is transferred such that all data bytes for the signal (including all elements of a wide signal) are from the same time step. Additionally, it is ensured that the relative sample time (delay) from which the data is transferred from one rate to another is always the same. Only ANSI-C code is used, no target specific 'critical section' protection is needed.

Ensure integrity (IntegOnly) : Data is transferred such that all data bytes for the signal (including all elements of a wide signal) are from the same time step. However, from one transfer of data to the next, the relative sample time (delay) for which the data is transferred can vary. In this mode, the code to read/write the data is run more often than in the DetandInt mode. In the worst case, the delay is equivalent to the DetandInt mode, but the delay can be less which is important is some applications. Also, this mode support data transfers to/from asynchronous rates which the DetandInt mode cannot support. Only ANSI-C code is used, no target specific 'critical section' protection is needed.

No data consistency operations are performed (None) : For this case, the Rate Transition block does not generated code. This mode is acceptable in some application where atomic access of scalar data types is guaranteed and when the relative temporal values of the data is not important. This mode does not introduce any delay.

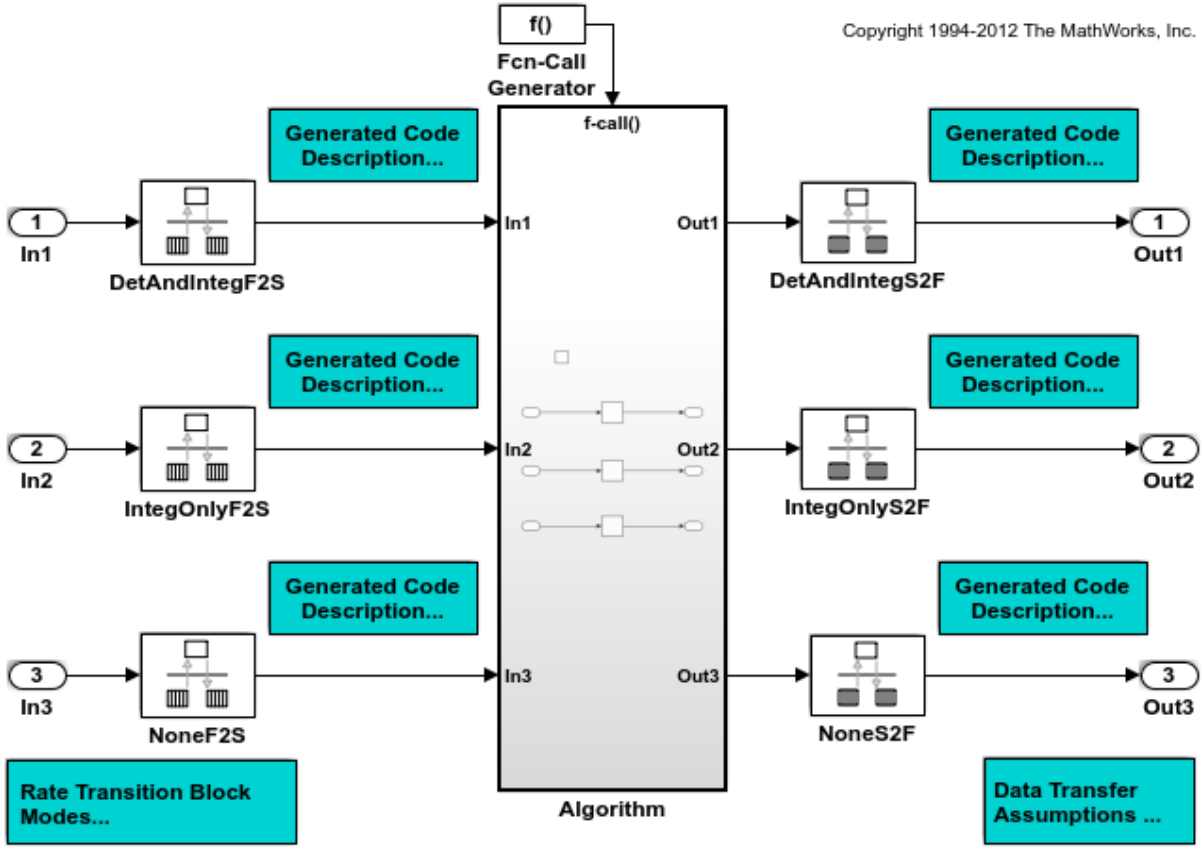
Data Transfer Assumptions

Basis of operation for data transfers between tasks:

- Data transitions occur between a single reading task and a single writing task.
- A read or write of a byte sized variable is atomic.
- When two tasks interact through a data transition, only one of them can preempt the other.
- For periodic tasks, the faster rate task has higher priority than the slower rate task; the faster rate task always preempts the slower rate task.
- All tasks run on a single processor. Time slicing is not allowed.
- Processes do not crash/restart (especially while data is being transferred between tasks)

Model `rtwdemo_ratetrans`

```
open_system('rtwdemo_ratetrans')
```



This model shows the differences in the operation modes of the Rate Transition block when used in a multirate, multitasking model. The flexible options for the Rate Transition block allow you to select the mode that is best suited for your application. You can trade levels of determinism and data integrity to improve system performance.

Generate Code Using Simulink Coder (double-click)

Generate Code Using Embedded Coder (double-click)

Display Sample Time Colors (double-click)

Model `rtwdemo_ratetrans` shows the differences in the operation modes of the following Rate Transition blocks.

Rate Transition block `DetAndIntegF2S`

Determinism and data integrity (fast to slow transition):

- The block output is used as a persistent data buffer.
- Data is written to output at slower rate but done during the faster rate context
- Data as seen by the slower rate is always the value when both the faster and slower rate last executed. Any subsequent steps by the faster rate (and associated data updates) while the slower rate is running are not seen by the slower rate.

Rate Transition block `DetAndIntegS2F`

Determinism and data integrity (slow to fast transition):

- Uses two persistent data buffers, an internal buffer and the blocks output.
- The internal buffer is copied to the output at the slower rate but done during the faster rate context.
- The internal buffer is written at the slower rate and during the slower rate context.
- The data that Fast rate sees is always delayed, i.e. data is from the previous step of the slow rate code.

Rate Transition block `IntegOnlyF2S`

Data integrity only (fast to slow transition):

- The block output is used as a persistent data buffer.
- Data is written to buffer during the faster rate context if a flag indicates it not in the process of being read.
- The flag is set and data is copied from the buffer to output at the slow rate, the flag is then cleared. This is an additional copy as compared to the deterministic case.
- Data as seen by the slower rate can be from a more recent step of the faster rate than from when the slower rate and faster rate both executed.

Rate Transition block `IntegOnlyS2F`

Data integrity only (slow to fast transition):

- Uses two persistent data buffers, both are internal buffers.

- One of the 2 buffers is always copied to the output at faster rate.
- One of the 2 buffers is written at the slower rate and during the slower rate context, then the active buffer is switched.
- The data as seen by the faster rate can be more recent than for the deterministic case. Specifically, when both the slower and faster rate have their hits, the faster rate will see a previous value from the slower rate. But, subsequent steps for the faster rate may see an updated value (when the slower rate updates the non-active buffer and switches the active buffer flag.

Rate Transition block `NoneF2S`

No code is generated for the Rate Transition block when determinism and data integrity is waived.

Rate Transition block `NoneS2F`

No code is generated for the Rate Transition block when determinism and data integrity is waived.

```
bdclose('rtwdemo_ratetrans');
```

More About

- “Time-Based Scheduling and Code Generation” (Simulink Coder)
- “Modeling for Single-Tasking Execution” (Simulink Coder)
- “Modeling for Multitasking Execution” (Simulink Coder)

Event-Based Scheduling in Simulink Coder

Asynchronous Events

Asynchronous Support

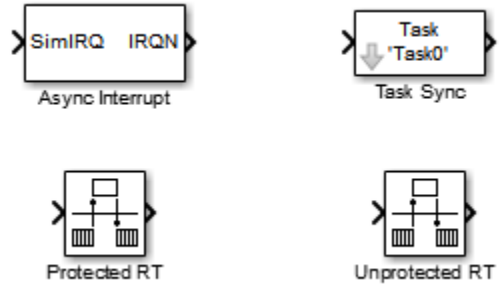
Normally, you time models from which you plan to generate code from a *periodic* interrupt source (for example, a hardware timer). Blocks in a periodically clocked single-rate model run at a timer interrupt rate (the base rate of the model). Blocks in a periodically clocked multirate model run at the base rate or at multiples of that rate.

Many systems must also support execution of blocks in response to events that are *asynchronous* with respect to the periodic timing source of the system. For example, a peripheral device might signal completion of an input operation by generating an interrupt. The system must service such interrupts, for example, by acquiring data from the interrupting device.

This chapter explains how to use blocks to model and generate code for asynchronous event handling, including servicing of hardware-generated interrupts, maintenance of timers, asynchronous read and write operations, and spawning of asynchronous tasks under a real-time operating system (RTOS). This block library demonstrates integration with an example RTOS (VxWorks). Although the blocks target an example RTOS, this chapter provides source code analysis and other information you can use to develop blocks that support asynchronous event handling for an alternative target RTOS.[4]

Block Library for Calls to an Example Real-Time Operating System

The next figure shows the blocks in the `vxlib1` block library.



4. VxWorks is a registered trademark of Wind River Systems, Inc.

The key blocks in the library are the Async Interrupt and Task Sync blocks. These blocks are targeted for an example RTOS (VxWorks). You can use them, with modification, to support your RTOS applications.

Note: You can use the blocks in the `vxlib1` (Simulink Coder) library (Async Interrupt and Task Sync) for simulation and code generation. These blocks provide starting point examples to help you develop custom blocks for your target environment.

To implement asynchronous support for an RTOS other than the example RTOS, use the guidelines and example code are provided to help you adapt the `vxlib1` library blocks to target your RTOS. This topic is discussed in "Create a Customized Asynchronous Library" on page 17-47.

The `vxlib1` library includes blocks you can use to

- Generate interrupt-level code — Async Interrupt block
- Spawn an RTOS task that calls a function call subsystem — Task Sync block
- Enable data integrity when transferring data between blocks running as different tasks — Protected RT block
- Use an unprotected/nondeterministic mode when transferring data between blocks running as different tasks — Unprotected RT block

The use of protected and unprotected Rate Transition blocks in asynchronous contexts is discussed in "Rate Transitions and Asynchronous Blocks" on page 17-39. For general information on rate transitions, see "Time-Based Scheduling and Code Generation" on page 16-2.

Access the Block Library for RTOS Integration

To access the example RTOS (VxWorks) block library, enter the MATLAB command `vxlib1`.

Generate Code Using Library Blocks for RTOS Integration

To generate an example RTOS compatible application from a model containing `vxlib1` library blocks, use the following configuration parameter values for your model.

- Select system target file `ert.tlc` (requires an Embedded Coder license) from the browse menu for the **Code Generation** > **System target file** parameter (`SystemTargetFile`).
- Enable the **Code Generation** > **Generate code only** parameter (`GenCodeOnly`).
- Enable the **All Parameters** > **Generate an example main program** parameter (`GenerateSampleERTMain`).
- Select `VxWorksExample` from the menu for the **All Parameters** > **Target operating system** parameter (`TargetOS`).

Examples and Additional Information

Additional information relevant to the topics in this chapter can be found in

- The `rtwdemo_async` model, which uses the `tornado.tlc` system target file and `vxlib1` block library. To open this example, type `rtwdemo_async` at the MATLAB command prompt.
- The `rtwdemo_async_mdlreftop` model, which uses the `tornado.tlc` system target file and `vxlib1` block library. To open this example, type `rtwdemo_async_mdlreftop` at the MATLAB command prompt.
- "Time-Based Scheduling and Code Generation" (Simulink Coder), discusses general multitasking and rate transition issues for periodic models.
- The Embedded Coder documentation discusses the `ert.tlc` system target file, including task execution and scheduling.
- For detailed information about the system calls to the example RTOS (VxWorks) mentioned in this chapter, see VxWorks system documentation on the Wind River website.

More About

- "Time-Based Scheduling and Code Generation" (Simulink Coder)
- "Generate Interrupt Service Routines" (Simulink Coder)
- "Spawn and Synchronize Execution of RTOS Task" (Simulink Coder)
- "Pass Asynchronous Events in RTOS as Input To a Referenced Model" (Simulink Coder)
- "Timers in Asynchronous Tasks" (Simulink Coder)
- "Import Asynchronous Event Data for Simulation" (Simulink Coder)

- “Rate Transitions and Asynchronous Blocks” (Simulink Coder)
- “Create a Customized Asynchronous Library” (Simulink Coder)
- “Asynchronous Support Limitations” (Simulink Coder)

Generate Interrupt Service Routines

To generate an interrupt service routine (ISR) associated with a specific VME interrupt level for the example RTOS (VxWorks), use the Async Interrupt block. The Async Interrupt block enables the specified interrupt level and installs an ISR that calls a connected function call subsystem.

You can also use the Async Interrupt block in a simulation. It provides an input port that can be enabled and connected to a simulated interrupt source.

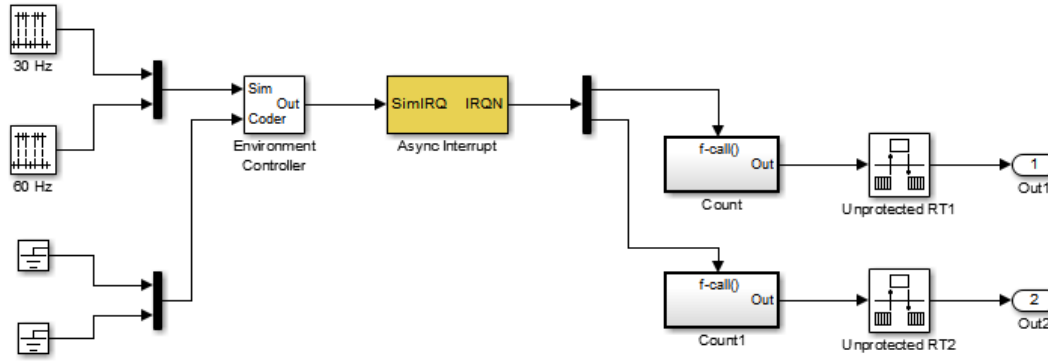
Note: The operating system integration techniques that are demonstrated in this section use one or more blocks the blocks in the `vxlib1` (Simulink Coder) library. These blocks provide starting point examples to help you develop custom blocks for your target environment.

Connecting the Async Interrupt Block

To generate an ISR, connect an output of the Async Interrupt block to the control input of

- A function call subsystem
- The input of a Task Sync block
- The input to a Stateflow chart configured for a function call input event

The next figure shows an Async Interrupt block configured to service two interrupt sources. The outputs (signal width 2) are connected to two function call subsystems.



Requirements and Restrictions

Note the following requirements and restrictions:

- The Async Interrupt block supports VME interrupts 1 through 7.
- The Async Interrupt block uses the following system calls to the example RTOS (VxWorks):
  - `sysIntEnable`
  - `sysIntDisable`
  - `intConnect`
  - `intLock`
  - `intUnlock`
  - `tickGet`

Performance Considerations

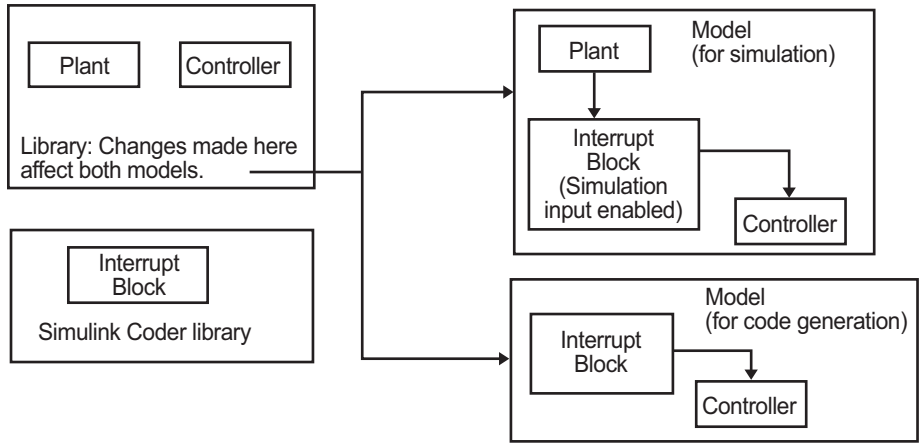
Execution of large subsystems at interrupt level can have a significant impact on interrupt response time for interrupts of equal and lower priority in the system. As a general rule, it is best to keep ISRs as short as possible. Connect only function call subsystems that contain a small number of blocks to an Async Interrupt block.

A better solution for large subsystems is to use the Task Sync block to synchronize the execution of the function call subsystem to a RTOS task. The Task Sync block is placed

between the Async Interrupt block and the function call subsystem. The Async Interrupt block then installs the Task Sync block as the ISR. The ISR releases a synchronization semaphore (performs a `semGive`) to the task, and returns immediately from interrupt level. The task is then scheduled and run by the example RTOS (VxWorks). See "Spawn and Synchronize Execution of RTOS Task" on page 17-15 for more information.

Using the Async Interrupt Block in Simulation and Code Generation

This section describes a *dual-model* approach to the development and implementation of real-time systems that include ISRs. In this approach, you develop one model that includes a plant and a controller for simulation, and another model that only includes the controller for code generation. Using a Simulink library, you can implement changes to both models simultaneously. The next figure shows how changes made to the plant or controller, both of which are in a library, are propagated to the models.

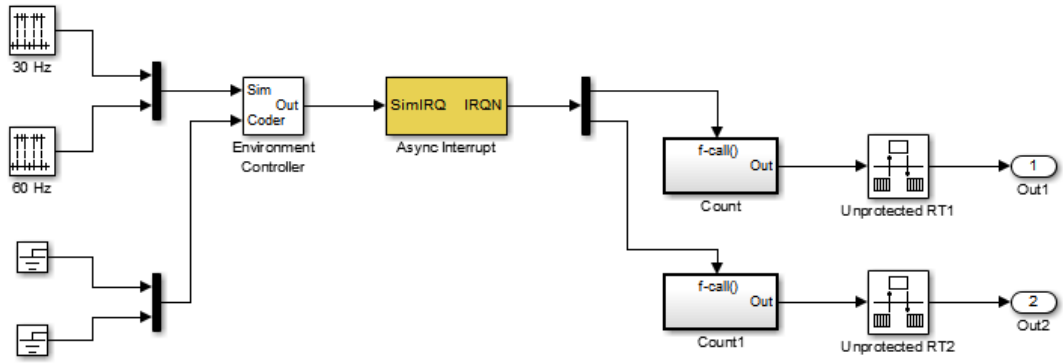


Dual-Model Use of Async Interrupt Block for Simulation and Code Generation

A *single-model* approach is also possible. In this approach, the Plant component of the model is active only in simulation. During code generation, the Plant components are effectively switched out of the system and code is generated only for the interrupt block and controller parts of the model. For an example of this approach, see the `rtwdemo_async` model.

Dual-Model Approach: Simulation

The following block diagram shows a simple model that illustrates the dual-model approach to modeling. During simulation, the Pulse Generator blocks provide simulated interrupt signals.



The simulated interrupt signals are routed through the Async Interrupt block's input port. Upon receiving a simulated interrupt, the block calls the connected subsystem.

During simulation, subsystems connected to Async Interrupt block outputs are executed in order of their priority in the example RTOS (VxWorks). In the event that two or more interrupt signals occur simultaneously, the Async Interrupt block executes the downstream systems in the order specified by their interrupt levels (level 7 gets the highest priority). The first input element maps to the first output element.

You can also use the Async Interrupt block in a simulation without enabling the simulation input. In such a case, the Async Interrupt block inherits the base rate of the model and calls the connected subsystems in order of their priorities in the RTOS. (In this case, the Async Interrupt block behaves as if all inputs received a 1 simultaneously.)

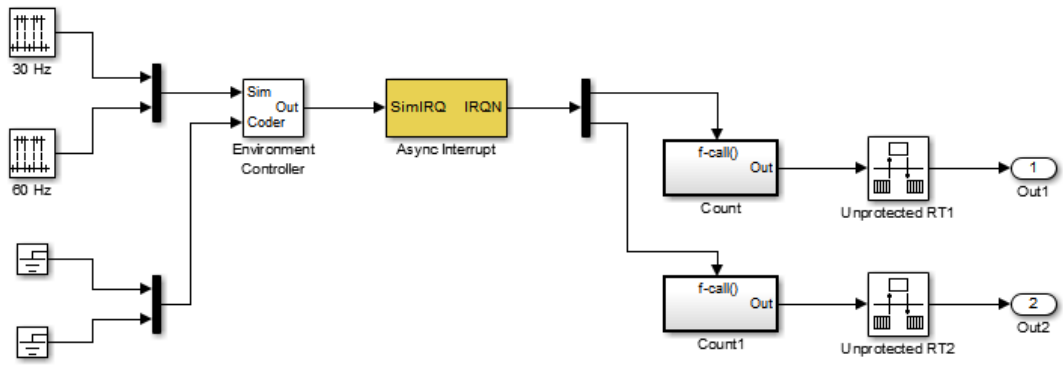
Dual-Model Approach: Code Generation

In the generated code for the sample model,

- Ground blocks provide input signals to the Environment Controller block

- The Async Interrupt block does not use its simulation input

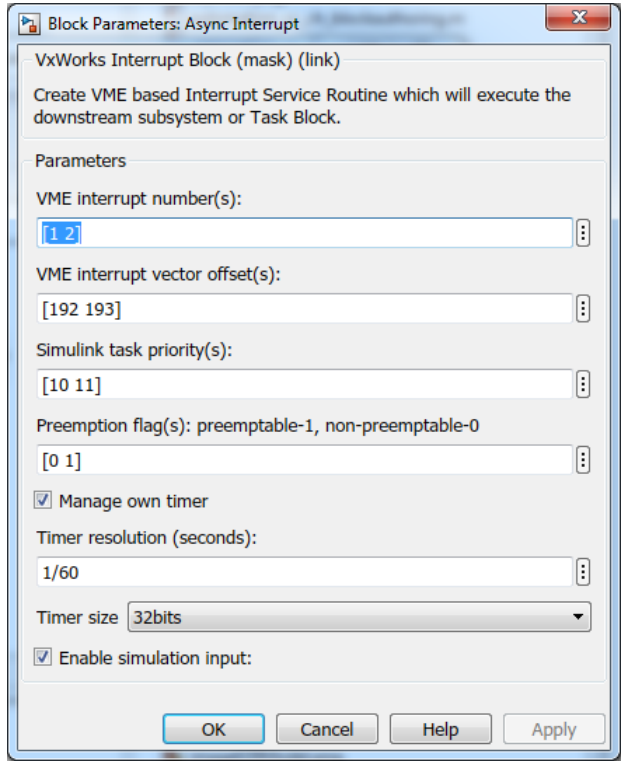
The Ground blocks drive control input of the Environment Controller block, so code is not generated for that signal path. The code generator does not produce code for blocks that drive the simulation control input to the Environment Controller block because that path is not selected during code generation. However, the sample times of driving blocks for the simulation input to the Environment Controller block contribute to the sample times supported in the generated code. To avoid including unnecessary sample times in the generated code, use the sample times of the blocks driving the simulation input in the model where generated code is intended.



Standalone functions are installed as ISRs and the interrupt vector table is as follows:

Offset	
192	`&isr_num1_vec192()`
193	`&isr_num2_vec193()`

Consider the code generated from this model, assuming that the Async Interrupt block parameters are configured as shown in the next figure.



Initialization Code

In the generated code, the Async Interrupt block installs the code in the Subsystem blocks as interrupt service routines. The interrupt vectors for `IRQ1` and `IRQ2` are stored at locations `192` and `193` relative to the base of the interrupt vector table, as specified by the **VME interrupt vector offset(s)** parameter.

Installing an ISR requires two RTOS (VxWorks) calls, `int_connect` and `sysInt_Enable`. The Async Interrupt block inserts these calls in the *`model`*`_initialize` function, as shown in the following code excerpt.

```
/* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
    /* Connect and enable ISR function: isr_num1_vec192 */
    if( intConnect(INUM_TO_IVEC(192), isr_num1_vec192, 0) != OK) {
      printf("intConnect failed for ISR 1.\n");
    }
    sysIntEnable(1);
```

```
/* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
/* Connect and enable ISR function: isr_num2_vec193 */
if( intConnect(INUM_TO_IVEC(193), isr_num2_vec193, 0) != OK)
{
  printf("intConnect failed for ISR 2.\n");
}
sysIntEnable(2);
```

The hardware that generates the interrupt is not configured by the Async Interrupt block. Typically, the interrupt source is a VME I/O board, which generates interrupts for specific events (for example, end of A/D conversion). The VME interrupt level and vector are set up in registers or by using jumpers on the board. You can use the `mdlStart` routine of a user-written device driver (S-function) to set up the registers and enable interrupt generation on the board. You must match the interrupt level and vector specified in the Async Interrupt block dialog to the level and vector set up on the I/O board.

Generated ISR Code

The actual ISR generated for `IRQ1` in the RTOS (VxWorks) is listed below.

```
/* VxWorks Interrupt Block: '<Root>/Async Interrupt' */

void isr_num1_vec192(void)
{
  int_T lock;
  FP_CONTEXT context;

   /* Use tickGet() as a portable tick counter example.
      A much higher resolution can be achieved with a
      hardware counter */
   Async_Code_M->Timing.clockTick2 = tickGet();

   /* disable interrupts (system is configured as non-ive) */
   lock = intLock();

   /* save floating point context */
   fppSave(&context);

   /* Call the system: <Root>/Subsystem A */
   Count(0, 0);

   /* restore floating point context */
   fppRestore(&context);
```

```
   /* re-enable interrupts */
   intUnlock(lock);
}
```

There are several features of the ISR that should be noted:

- Because of the setting of the **Preemption Flag(s)** parameter, this ISR is locked; that is, it cannot be preempted by a higher priority interrupt. The ISR is locked and unlocked in the example RTOS (VxWorks) by the `int_lock` and `int_unlock` functions.
- The connected subsystem, `Count`, is called from within the ISR.
- The `Count` function executes algorithmic (model) code. Therefore, the floating-point context is saved and restored across the call to `Count`.
- The ISR maintains its own absolute time counter, which is distinct from other periodic base rate or subrate counters in the system. Timing data is maintained for the use of any blocks executed within the ISR that require absolute or elapsed time.

  See “Timers in Asynchronous Tasks” on page 17-44 for details.

Model Termination Code

The model's termination function disables the interrupts in the RTOS (VxWorks):

```
/* Model terminate function */
void Async_Code_terminate(void)
{
   /* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
   /* Disable interrupt for ISR system: isr_num1_vec192 */
   sysIntDisable(1);

   /* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
   /* Disable interrupt for ISR system: isr_num2_vec193 */
   sysIntDisable(2);
}
```

More About

- “Spawn and Synchronize Execution of RTOS Task” (Simulink Coder)
- “Pass Asynchronous Events in RTOS as Input To a Referenced Model” (Simulink Coder)
- “Import Asynchronous Event Data for Simulation” (Simulink Coder)

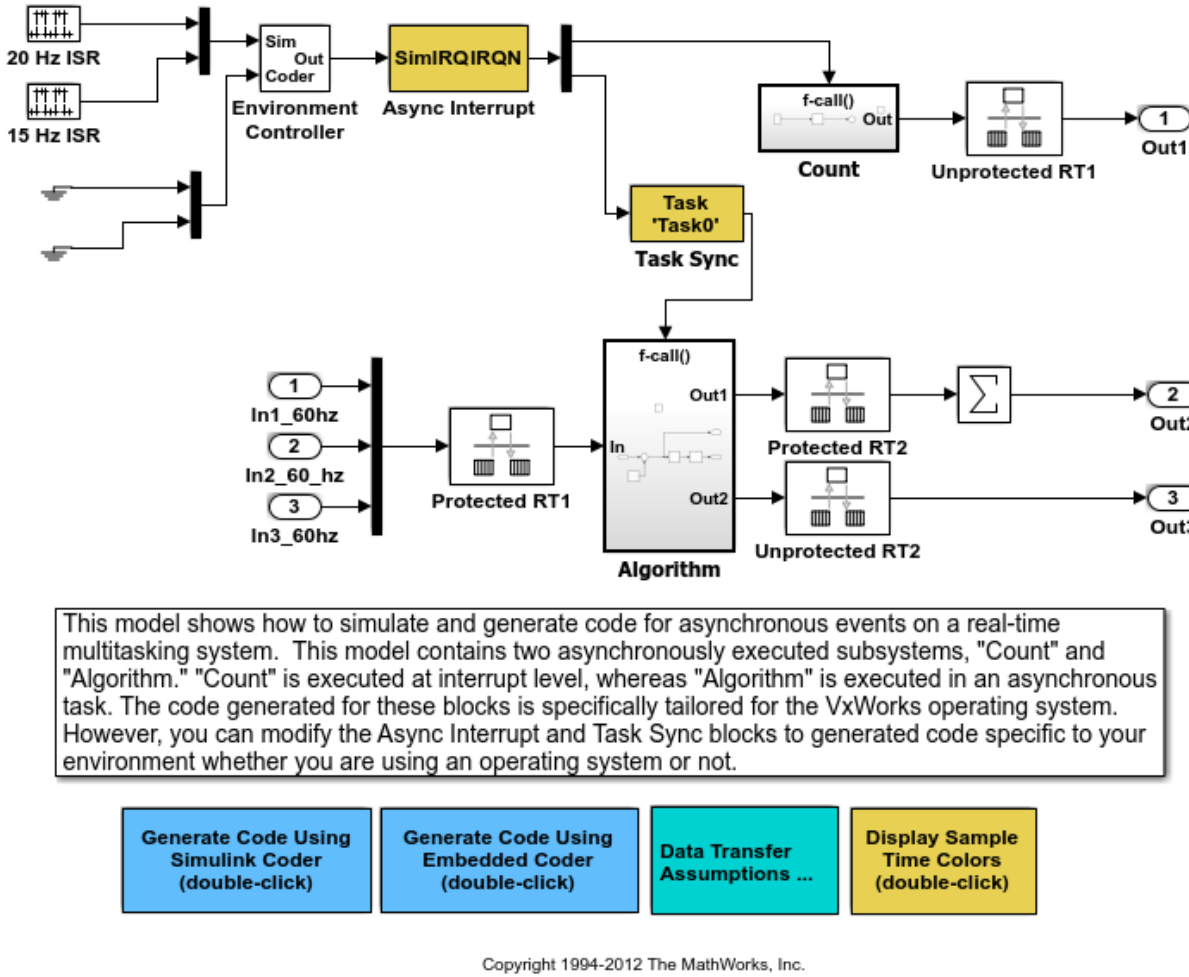
- “Rate Transitions and Asynchronous Blocks” (Simulink Coder)

Spawn and Synchronize Execution of RTOS Task

This example shows how to simulate and generate code for asynchronous events on a multitasking real-time operating system (VxWorks®). The model shows different techniques for handling asynchronous events depending on the size of the triggered subsystems.

About the Example Model

Open the example model `rtwdemo_async`.



The model simulates an interrupt source and includes an Async Interrupt block, a Task Sync block, function-call subsystems `Count` and `Algorithm`, and Rate Transition blocks. The Async Interrupt block creates two Versa Module Eurocard (VME) interrupt service routines (ISRs) that pass interrupt signals to subsystem `Count` and the Task Sync block. You can place an Async Interrupt block between a simulated interrupt source and one of the following:

- Function call subsystem
- Task Sync block
- A Stateflow® chart configured for a function call input event
- A referenced model with an Inport block that connects to one of the preceding model elements

The Async Interrupt and Task Sync blocks enable the subsystems to execute asynchronously.

`Count` represents a simple interrupt service routine (ISR) that executes at interrupt level. It is best to keep ISRs as simple as possible. This subsystem includes only a Discrete-Time Integrator block.

`Algorithm` includes more substance. It includes multiple blocks and produces two output values. Execution of larger subsystems at interrupt level can significantly impact response time for interrupts of equal and lower priority in the system. A better solution for larger subsystems is to use the Task Sync block to represent the ISR for the function-call subsystem.

The Async Interrupt block generates calls to ISRs. Place the block between a simulated interrupt source and one of the following:

- Function call subsystem
- Task Sync block
- A Stateflow® chart configured for a function call input event

For each specified interrupt level, the block generates a Versa Module Eurocard (VME) ISR that executes the connected subsystem, Task Sync block, or chart.

In the example model, the Async Interrupt block is configured for VME interrupts 1 and 2, by using interrupt vector offsets 192 and 193. Interrupt 1 connects directly to subsystem `Count`. Interrupt 2 connects to a Task Sync block, which serves as the ISR for `Algorithm`. Place a Task Sync block in one of the following locations:

- Between an Async Interrupt block and a function-call subsystem or Stateflow® chart.
- At the output port of a Stateflow® chart that has an event, `Output to Simulink`, that you configure as a function call.

In the example model, the Task Sync block is between the Async Interrupt block and function-call subsystem `Algorithm`. The Task Sync block is configured with the task name `Task()`, a priority of 50, a stack size of 8192, and data transfers of the task

synchronized with the caller task. The spawned task uses a semaphore to synchronize task execution. The Async Interrupt block triggers a release of the task semaphore.

Four Rate Transition blocks handle data transfers between ports that operate at different rates. In two instances, Protected Rate Transition blocks protect data transfers (prevent them from being preempted and corrupted). In the other two instances, Unprotected Rate Transition blocks introduce no special behavior. Their presence informs Simulink® of a rate transition.

The code generated for the Async Interrupt and Task Sync blocks is tailored for the example RTOS (VxWorks®). However, you can modify the blocks to generate code specific to your run-time environment.

Data Transfer Assumptions

- Data transfers occur between one reading task and one writing task.
- A read or write operation on a byte-size variable is atomic.
- When two tasks interact, only one can preempt the other.
- For periodic tasks, the task with the faster rate has higher priority than the task with the slower rate. The task with the faster rate preempts the tasks with slower rates.
- Tasks run on a single processor. Time slicing is not allowed.
- Processes do not stop and restart, especially while data is being transferred between tasks.

Simulate the Model

Simulate the model. By default, the model is configured to show sample times in different colors. Discrete sample times for input and output appear red and green, respectively. Constants are reddish-blue. Asynchronous interrupts and tasks are purple. The Rate Transition Blocks, which are a hybrid rate (their input and output sample times can differ), are yellow.

Generate Code and Report

Generate code and a code generation report for the model. Generated code for the Async Interrupt and Task Sync blocks is for the example RTOS (VxWorks®). However, you can modify the blocks to generate code for another run-time environment.

1. Create a temporary folder for the build and inspection process.

2. Build the model.

```
### Starting build procedure for model: rtwdemo_async
Warning:  Simulink Coder: The tornado.tlc target will be removed in a future release.

### Wrapping unrecognized make command (angle brackets added)
###    <make>
### in default batch file
### Successful completion of code generation for model: rtwdemo_async
```

Review Initialization Code

Open the generated source file `rtwdemo_async.c`. The initialization code:

1. Creates and initializes the synchronization semaphore `Task0_semaphore`.

```
*(SEM_ID *)rtwdemo_async_DW.SFunction_PWORK.SemID = semBCreate(SEM_Q_PRIORITY,
  SEM_EMPTY);
if (rtwdemo_async_DW.SFunction_PWORK.SemID == NULL) {
  printf("semBCreate call failed for block Task0.\n");
}
```

2. Spawns task `task0` and assigns the task priority 50.

```
rtwdemo_async_DW.SFunction_IWORK.TaskID = taskSpawn("Task0",
  50.0,
  VX_FP_TASK,
  8192.0,
  (FUNCPTR)Task0,
  0, 0, 0, 0, 0, 0, 0,0, 0, 0);
if (rtwdemo_async_DW.SFunction_IWORK.TaskID == ERROR) {
  printf("taskSpawn call failed for block Task0.\n");
}

/* End of Start for S-Function (vxtask1): '<S5>/S-Function' */

/* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
/* Connect and enable ISR function: isr_num1_vec192 */
if (intConnect(INUM_TO_IVEC(192), isr_num1_vec192, 0) != OK) {
  printf("intConnect failed for ISR 1.\n");
}

sysIntEnable(1);

/* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
```

```
/* Connect and enable ISR function: isr_num2_vec193 */
if (intConnect(INUM_TO_IVEC(193), isr_num2_vec193, 0) != OK) {
  printf("intConnect failed for ISR 2.\n");
}

sysIntEnable(2);
```

3. Connects and enables ISR `isr_num1_vec192` for interrupt 1 and ISR `isr_num2_vec193` for interrupt 2.

```
{
  int32_T i;
  for (i = 0; i < 60; i++) {
    /* InitializeConditions for RateTransition: '<Root>/Protected RT1' */
    rtwdemo_async_DW.ProtectedRT1_Buffer[i] = 0.0;

    /* InitializeConditions for RateTransition: '<Root>/Protected RT2' */
    rtwdemo_async_DW.ProtectedRT2_Buffer[i] = 0.0;
  }

  /* SystemInitialize for S-Function (vxinterrupt1): '<Root>/Async Interrupt' incorpo
   *  SystemInitialize for SubSystem: '<Root>/Count'
   */
  /* System initialize for function-call system: '<Root>/Count' */
  rtwdemo_async_DW.Count_PREV_T = rtwdemo_async_M->Timing.clockTick2;

  /* InitializeConditions for DiscreteIntegrator: '<S2>/Integrator' */
  rtwdemo_async_DW.Integrator_DSTATE_l = 0.0;

  /* SystemInitialize for Outport: '<Root>/Out1' incorporates:
   *  SystemInitialize for Outport: '<S2>/Out'
   */
  rtwdemo_async_Y.Out1 = 0.0;

  /* SystemInitialize for S-Function (vxinterrupt1): '<Root>/Async Interrupt' incorpo
   *  SystemInitialize for SubSystem: '<S4>/Subsystem'
   */

  /* System initialize for function-call system: '<S4>/Subsystem' */

  /* SystemInitialize for S-Function (vxtask1): '<S5>/S-Function' incorporates:
   *  SystemInitialize for SubSystem: '<Root>/Algorithm'
   */
```

```
    /* System initialize for function-call system: '<Root>/Algorithm' */
    rtwdemo_async_M->Timing.clockTick4 = rtwdemo_async_M->Timing.clockTick3;
    rtwdemo_async_DW.Algorithm_PREV_T = rtwdemo_async_M->Timing.clockTick4;

    /* InitializeConditions for DiscreteIntegrator: '<S1>/Integrator' */
    rtwdemo_async_DW.Integrator_DSTATE = 0.0;

    /* SystemInitialize for Outport: '<S1>/Out1' */
    memset(&rtwdemo_async_B.Sum[0], 0, 60U * sizeof(real_T));

    /* SystemInitialize for Outport: '<Root>/Out3' incorporates:
     *  SystemInitialize for Outport: '<S1>/Out2'
     */
    rtwdemo_async_Y.Out3 = 0.0;

    /* End of SystemInitialize for S-Function (vxtask1): '<S5>/S-Function' */

    /* End of SystemInitialize for S-Function (vxinterrupt1): '<Root>/Async Interrupt'
  }
}

/* Model terminate function */
static void rtwdemo_async_terminate(void)
{
  /* Terminate for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */

  /* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
  /* Disable interrupt for ISR system: isr_num1_vec192 */
  sysIntDisable(1);

  /* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
  /* Disable interrupt for ISR system: isr_num2_vec193 */
  sysIntDisable(2);

  /* End of Terminate for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */

  /* Terminate for S-Function (vxinterrupt1): '<Root>/Async Interrupt' incorporates:
   *  Terminate for SubSystem: '<S4>/Subsystem'
   */

  /* Termination for function-call system: '<S4>/Subsystem' */

  /* Terminate for S-Function (vxtask1): '<S5>/S-Function' */
```

```
  /* VxWorks Task Block: '<S5>/S-Function' (vxtask1) */
  /* Destroy task: Task0 */
  taskDelete(rtwdemo_async_DW.SFunction_IWORK.TaskID);

  /* End of Terminate for S-Function (vxtask1): '<S5>/S-Function' */

  /* End of Terminate for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */
}

/*========================================================================*
 * Start of Classic call interface                                        *
 *========================================================================*/
void MdlOutputs(int_T tid)
{
  rtwdemo_async_output(tid);
}

void MdlUpdate(int_T tid)
{
  rtwdemo_async_update(tid);
}

void MdlInitializeSizes(void)
{
}

void MdlInitializeSampleTimes(void)
{
}

void MdlInitialize(void)
{
}

void MdlStart(void)
{
  rtwdemo_async_initialize();
}

void MdlTerminate(void)
{
  rtwdemo_async_terminate();
}
```

```
/* Registration function */
RT_MODEL_rtwdemo_async_T *rtwdemo_async(void)
{
  /* Registration code */

  /* initialize non-finites */
  rt_InitInfAndNaN(sizeof(real_T));

  /* initialize real-time model */
  (void) memset((void *)rtwdemo_async_M, 0,
                sizeof(RT_MODEL_rtwdemo_async_T));

  /* Initialize timing info */
  {
    int_T *mdlTsMap = rtwdemo_async_M->Timing.sampleTimeTaskIDArray;
    mdlTsMap[0] = 0;
    mdlTsMap[1] = 1;
    rtwdemo_async_M->Timing.sampleTimeTaskIDPtr = (&mdlTsMap[0]);
    rtwdemo_async_M->Timing.sampleTimes =
      (&rtwdemo_async_M->Timing.sampleTimesArray[0]);
    rtwdemo_async_M->Timing.offsetTimes =
      (&rtwdemo_async_M->Timing.offsetTimesArray[0]);

    /* task periods */
    rtwdemo_async_M->Timing.sampleTimes[0] = (0.016666666666666666);
    rtwdemo_async_M->Timing.sampleTimes[1] = (0.05);

    /* task offsets */
    rtwdemo_async_M->Timing.offsetTimes[0] = (0.0);
    rtwdemo_async_M->Timing.offsetTimes[1] = (0.0);
  }

  rtmSetTPtr(rtwdemo_async_M, &rtwdemo_async_M->Timing.tArray[0]);

  {
    int_T *mdlSampleHits = rtwdemo_async_M->Timing.sampleHitArray;
    int_T *mdlPerTaskSampleHits = rtwdemo_async_M->Timing.perTaskSampleHitsArray;
    rtwdemo_async_M->Timing.perTaskSampleHits = (&mdlPerTaskSampleHits[0]);
    mdlSampleHits[0] = 1;
    rtwdemo_async_M->Timing.sampleHits = (&mdlSampleHits[0]);
  }

  rtmSetTFinal(rtwdemo_async_M, 0.5);
  rtwdemo_async_M->Timing.stepSize0 = 0.016666666666666666;
```

```
  rtwdemo_async_M->Timing.stepSize1 = 0.05;
  rtwdemo_async_M->solverInfoPtr = (&rtwdemo_async_M->solverInfo);
  rtwdemo_async_M->Timing.stepSize = (0.016666666666666666);
  rtsiSetFixedStepSize(&rtwdemo_async_M->solverInfo, 0.016666666666666666);
  rtsiSetSolverMode(&rtwdemo_async_M->solverInfo, SOLVER_MODE_MULTITASKING);

  /* block I/O */
  rtwdemo_async_M->blockIO = ((void *) &rtwdemo_async_B);
  (void) memset(((void *) &rtwdemo_async_B), 0,
                sizeof(B_rtwdemo_async_T));

  /* states (dwork) */
  rtwdemo_async_M->dwork = ((void *) &rtwdemo_async_DW);
  (void) memset((void *)&rtwdemo_async_DW, 0,
                sizeof(DW_rtwdemo_async_T));

  /* external inputs */
  rtwdemo_async_M->inputs = (((void*)&rtwdemo_async_U));
  (void)memset((void *)&rtwdemo_async_U, 0, sizeof(ExtU_rtwdemo_async_T));

  /* external outputs */
  rtwdemo_async_M->outputs = (&rtwdemo_async_Y);
  (void) memset((void *)&rtwdemo_async_Y, 0,
                sizeof(ExtY_rtwdemo_async_T));

  /* Initialize Sizes */
  rtwdemo_async_M->Sizes.numContStates = (0);/* Number of continuous states */
  rtwdemo_async_M->Sizes.numY = (3);   /* Number of model outputs */
  rtwdemo_async_M->Sizes.numU = (60);  /* Number of model inputs */
  rtwdemo_async_M->Sizes.sysDirFeedThru = (0);/* The model is not direct feedthrough */
  rtwdemo_async_M->Sizes.numSampTimes = (2);/* Number of sample times */
  rtwdemo_async_M->Sizes.numBlocks = (17);/* Number of blocks */
  rtwdemo_async_M->Sizes.numBlockIO = (4);/* Number of block outputs */
  return rtwdemo_async_M;
}

/*========================================================================*
 * End of Classic call interface                                          *
 *========================================================================*/
```

The order of these operations is important. Before the code generator enables the interrupt that activates the task, it must spawn the task.

Review Task and Task Synchronization Code

In the generated source file `rtwdemo_async.c`, review the task and task synchronization code.

The code generator produces the code for function `Task0` from the Task Sync block. That function includes a small amount of interrupt-level code and runs as an RTOS task.

The task waits in an infinite `for` loop until the system releases a synchronization semaphore. If the system releases the semaphore, the function updates its task timer and calls the code generated for the `Algorithm` subsystem.

In the example model, the **Synchronize the data transfer of this task with the caller task** parameter for the Task Sync block is set. This parameter setting updates the timer associated with the Task Sync block (`rtM->Timing.clockTick2`) with the value of the timer that the Async Interrupt block (`rtM->Timing.clockTick3`) maintains. As a result, code for blocks within the `Algorithm` subsystem use timer values that are based on the time of the most recent interrupt, rather than the most recent activation of `Task0`.

```
{
  /* Wait for semaphore to be released by system: rtwdemo_async/Task Sync */
  for (;;) {
    if (semTake(*(SEM_ID *)rtwdemo_async_DW.SFunction_PWORK.SemID,NO_WAIT) !=
        ERROR) {
      logMsg("Rate for Task Task0() too fast.\n",0,0,0,0,0,0);

#if STOPONOVERRUN

      logMsg("Aborting real-time simulation.\n",0,0,0,0,0,0);
      semGive(stopSem);
      return(ERROR);

#endif

    } else {
      semTake(*(SEM_ID *)rtwdemo_async_DW.SFunction_PWORK.SemID, WAIT_FOREVER);
    }

    /* Use the upstream clock tick counter for this Task. */
    rtwdemo_async_M->Timing.clockTick4 = rtwdemo_async_M->Timing.clockTick3;

    /* Call the system: '<Root>/Algorithm' */
```

```
{
  {
    int32_T tmp;
    int32_T i;

    /* RateTransition: '<Root>/Protected RT1' */
    tmp = rtwdemo_async_DW.ProtectedRT1_ActiveBufIdx * 60;
    for (i = 0; i < 60; i++) {
      rtwdemo_async_B.ProtectedRT1[i] =
        rtwdemo_async_DW.ProtectedRT1_Buffer[i + tmp];
    }

    /* End of RateTransition: '<Root>/Protected RT1' */

    /* S-Function (vxinterrupt1): '<Root>/Async Interrupt' */

    /* S-Function (vxtask1): '<S5>/S-Function' */

    /* Output and update for function-call system: '<Root>/Algorithm' */
    {
      real_T tmp;
      int32_T i;
      rtwdemo_async_M->Timing.clockTick4 =
        rtwdemo_async_M->Timing.clockTick3;
      rtwdemo_async_DW.Algorithm_ELAPS_T =
        rtwdemo_async_M->Timing.clockTick4 -
        rtwdemo_async_DW.Algorithm_PREV_T;
      rtwdemo_async_DW.Algorithm_PREV_T = rtwdemo_async_M->Timing.clockTick4;

      /* Outport: '<Root>/Out3' incorporates:
       *  DiscreteIntegrator: '<S1>/Integrator'
       */
      rtwdemo_async_Y.Out3 = rtwdemo_async_DW.Integrator_DSTATE;

      /* Sum: '<S1>/Sum' incorporates:
       *  Constant: '<S1>/Offset'
       */
      rtwdemo_async_B.Sum[0] = rtwdemo_async_B.ProtectedRT1[0] + 1.25;

      /* Sum: '<S1>/Sum1' */
      tmp = rtwdemo_async_B.Sum[0];
      for (i = 0; i < 59; i++) {
        /* Sum: '<S1>/Sum' incorporates:
         *  Constant: '<S1>/Offset'
```

```
             */
            rtwdemo_async_B.Sum[i + 1] = rtwdemo_async_B.ProtectedRT1[i + 1] +
              1.25;

            /* Sum: '<S1>/Sum1' incorporates:
             *  Sum: '<S1>/Sum'
             */
            tmp += rtwdemo_async_B.Sum[i + 1];
          }

          /* Update for DiscreteIntegrator: '<S1>/Integrator' incorporates:
           *  Sum: '<S1>/Sum1'
           */
          rtwdemo_async_DW.Integrator_DSTATE += 0.016666666666666666 * (real_T)
            rtwdemo_async_DW.Algorithm_ELAPS_T * tmp;
        }

        /* End of Outputs for S-Function (vxtask1): '<S5>/S-Function' */

        /* End of Outputs for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */
      }

      {
        int32_T i;

        /* Update for RateTransition: '<Root>/Protected RT2' */
        for (i = 0; i < 60; i++) {
          rtwdemo_async_DW.ProtectedRT2_Buffer[i +
            (rtwdemo_async_DW.ProtectedRT2_ActiveBufIdx == 0) * 60] =
            rtwdemo_async_B.Sum[i];
        }

        rtwdemo_async_DW.ProtectedRT2_ActiveBufIdx = (int8_T)
          (rtwdemo_async_DW.ProtectedRT2_ActiveBufIdx == 0);

        /* End of Update for RateTransition: '<Root>/Protected RT2' */
      }
    }
  }
}
```

The code generator produces code for ISRs `isr_num1_vec192` and `isr_num2_vec293`. ISR `isr_num2_vec192`:

- Disables interrupts.
- Saves floating-point context.
- Calls the code generated for the subsystem that connects to the referenced model Inport block, which receives the interrupt.
- Restores floating-point context.
- Reenables interrupts.

```
void isr_num1_vec192(void)
{
  int_T lock;
  FP_CONTEXT context;

  /* Use tickGet()  as a portable tick
     counter example. A much higher resolution can
     be achieved with a hardware counter */
  rtwdemo_async_M->Timing.clockTick2 = tickGet();

  /* disable interrupts (system is configured as non-preemptive) */
  lock = intLock();

  /* save floating point context */
  fppSave(&context);

  /* Call the system: '<Root>/Count' */
  {
    /* S-Function (vxinterrupt1): '<Root>/Async Interrupt' */

    /* Output and update for function-call system: '<Root>/Count' */
    rtwdemo_async_DW.Count_ELAPS_T = rtwdemo_async_M->Timing.clockTick2 -
      rtwdemo_async_DW.Count_PREV_T;
    rtwdemo_async_DW.Count_PREV_T = rtwdemo_async_M->Timing.clockTick2;

    /* Outport: '<Root>/Out1' incorporates:
     *  DiscreteIntegrator: '<S2>/Integrator'
     */
    rtwdemo_async_Y.Out1 = rtwdemo_async_DW.Integrator_DSTATE_l;

    /* Update for DiscreteIntegrator: '<S2>/Integrator' */
    rtwdemo_async_DW.Integrator_DSTATE_l += 0.016666666666666666 * (real_T)
      rtwdemo_async_DW.Count_ELAPS_T;

    /* End of Outputs for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */
```

```
  }

  /* restore floating point context */
  fppRestore(&context);

  /* re-enable interrupts */
  intUnlock(lock);
}

/* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
```

ISR `isr_num2_vec293` maintains a timer that stores the tick count at the time that the interrupt occurs. After updating the timer, the ISR releases the semaphore that activates `Task0`.

```
void isr_num2_vec193(void)
{
  /* Use tickGet()  as a portable tick
     counter example. A much higher resolution can
     be achieved with a hardware counter */
  rtwdemo_async_M->Timing.clockTick3 = tickGet();

  /* Call the system: '<S4>/Subsystem' */
  {
    /* S-Function (vxinterrupt1): '<Root>/Async Interrupt' */

    /* Output and update for function-call system: '<S4>/Subsystem' */

    /* S-Function (vxtask1): '<S5>/S-Function' */

    /* VxWorks Task Block: '<S5>/S-Function' (vxtask1) */
    /* Release semaphore for system task: Task0 */
    semGive(*(SEM_ID *)rtwdemo_async_DW.SFunction_PWORK.SemID);

    /* End of Outputs for S-Function (vxtask1): '<S5>/S-Function' */

    /* End of Outputs for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */
  }
}

/* VxWorks Task Block: '<S5>/S-Function' (vxtask1) */
```

Review Task Termination Code

The Task Sync block generates the following termination code.

```
static void rtwdemo_async_terminate(void)
{
  /* Terminate for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */

  /* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
  /* Disable interrupt for ISR system: isr_num1_vec192 */
  sysIntDisable(1);

  /* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
  /* Disable interrupt for ISR system: isr_num2_vec193 */
  sysIntDisable(2);

  /* End of Terminate for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */

  /* Terminate for S-Function (vxinterrupt1): '<Root>/Async Interrupt' incorporates:
   *  Terminate for SubSystem: '<S4>/Subsystem'
   */

  /* Termination for function-call system: '<S4>/Subsystem' */

  /* Terminate for S-Function (vxtask1): '<S5>/S-Function' */

  /* VxWorks Task Block: '<S5>/S-Function' (vxtask1) */
  /* Destroy task: Task0 */
  taskDelete(rtwdemo_async_DW.SFunction_IWORK.TaskID);

  /* End of Terminate for S-Function (vxtask1): '<S5>/S-Function' */

  /* End of Terminate for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */
}
```

Related Information

- Async Interrupt (Simulink Coder)
- Task Sync (Simulink Coder)
- “Generate Interrupt Service Routines” (Simulink Coder)
- “Timers in Asynchronous Tasks” (Simulink Coder)
- “Create a Customized Asynchronous Library” (Simulink Coder)

- “Import Asynchronous Event Data for Simulation” (Simulink Coder)
- “Load Data to Root-Level Input Ports” (Simulink)
- “Asynchronous Events” (Simulink Coder)
- “Rate Transitions and Asynchronous Blocks” (Simulink Coder)
- “Asynchronous Support Limitations” (Simulink Coder)

More About

- “Generate Interrupt Service Routines” (Simulink Coder)
- “Pass Asynchronous Events in RTOS as Input To a Referenced Model” (Simulink Coder)
- “Timers in Asynchronous Tasks” (Simulink Coder)
- “Import Asynchronous Event Data for Simulation” (Simulink Coder)
- “Rate Transitions and Asynchronous Blocks” (Simulink Coder)

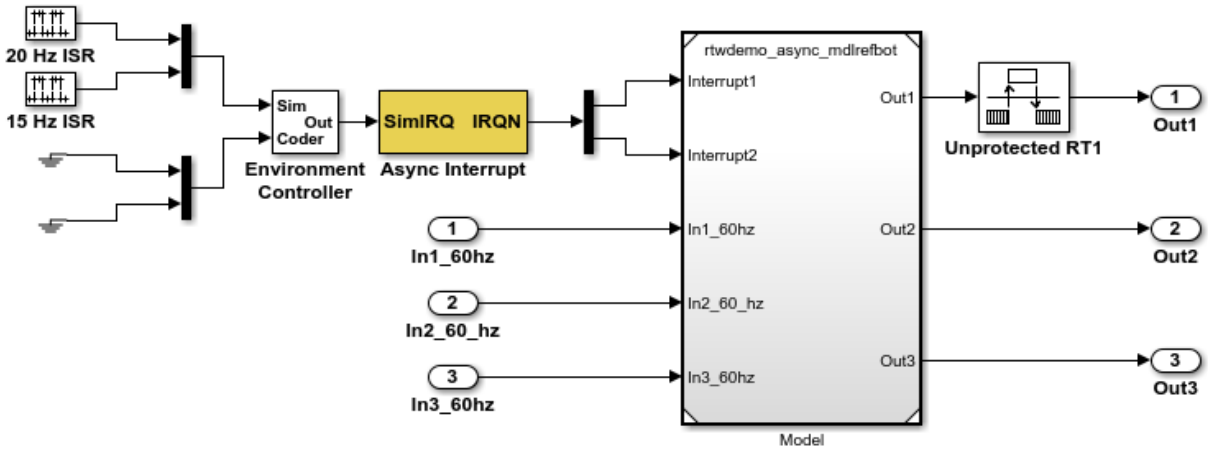
Pass Asynchronous Events in RTOS as Input To a Referenced Model

This example shows how to simulate and generate code for a model that triggers asynchronous events in an example RTOS (VxWorks®) that get passed as input to a referenced model.

Open Example Model

Open the example model `rtwdemo_async_mdlreftop`.

```
Warning: Undefined function 'LibraryBrowserCustomizer' for
input arguments of type 'DAStudio.CustomizationManager'.
```



This model shows how to simulate and generate code for asynchronous events on a real-time multitasking system. The two asynchronous events, "Interrupt1" and "Interrupt2", are executed in the referenced model via two different function-call input ports. The code generated for these blocks is specifically tailored for the VxWorks operating system. However, you can modify the Async Interrupt block to generate code specific to your environment whether or not you are using an operating system.

The model simulates an interrupt source and includes an Async Interrupt block and referenced model. The Async Interrupt block creates two Versa Module Eurocard (VME) interrupt service routines (ISRs) that pass interrupt signals to Inport blocks 1 and 2 of the referenced model. You can place an Async Interrupt block between a simulated interrupt source and one of the following:

- Function call subsystem
- Task Sync block
- A Stateflow® chart configured for a function call input event
- A referenced model with a Inport block that connects to one of the preceding model elements

In this example model, the Async Interrupt block passes asynchronous events (function-call trigger signals), `Interrupt1` and `Interrupt2`, to the referenced model through Inport blocks 1 and 2.

The code generated for the Async Interrupt block is tailored for the example real-time operating system (VxWorks). However, you can modify the block to generate code specific to your run-time environment.

Open the referenced model.

The referenced model includes the two Inport blocks that receive the interrupts, each connected to an Asynchronous Task Specification block, function-call subsystems `Count` and `Algorithm`, and Rate Transition blocks. The Asynchronous Task Specification block, in combination with a root-level Inport block, allows a reference model to receive asynchronous function-call input. To use the block:

1. Connect the Asynchronous Task Specification block to the output port of a root-level Inport block that outputs a function-call trigger.
2. Select the **Output function call** parameter of the Inport block to specify that it accepts function-call signals.
3. On the Asynchronous Task Specification parameters dialog box, set the task priority for the asynchronous task associated with an Inport block. Specify an integer or []. If you specify an integer, it must match the priority of the interrupt initiated by the Async Interrupt block in the parent model. If you specify [], the priorities do not have to match.

The Asynchronous Task Specification block for the higher priority interrupt, `interrupt1`, connects to function-call subsystem `Count`. `Count` represents a simple interrupt service routine (ISR). The second Asynchronous Task Specification block

connects to the subsystem `Algorithm`, which includes more substance. It includes multiple blocks and produces two output values. Both subsystems execute at interrupt level.

For each interrupt level specified for the Async Interrupt block in the parent model, the block generates a VME ISR that executes the connected subsystem, Task Sync block, or chart.

In the example top model, the Async Interrupt block is configured for VME interrupts 1 and 2, using interrupt vector offsets 192 and 193. Interrupt 1 is wired to trigger subsystem `Count`. Interrupt 2 is wired to trigger subsystem `Algorithm`.

The Rate Transition blocks handle data transfers between ports that operate at different rates. In two instances, the blocks protect data transfers (prevent them from being preempted and corrupted). In the other instance, no special behavior occurs.

Data Transfer Assumptions

- Data transfers occur between one reading task and one writing task.
- A read or write operation on a byte-sized variable is atomic.
- When two tasks interact, only one can preempt the other.
- For periodic tasks, the task with the faster rate has higher priority than the task with the slower rate. The task with the faster rate preempts the tasks slower rates.
- Tasks run on a single processor. Time slicing is not allowed.
- Processes do not crash and restart, especially while data is being transferred between tasks.

Simulate the Model

Simulate the model. By default, the model is configured to show sample times in different colors. Discrete sample times for input and output appear red and green, respectively. Constants are magenta. Asynchronous interrupts are purple. The Rate Transition Blocks, which are hybrid (input and output sample times can differ), appear yellow.

Generate Code and Report

Generate code and a code generation report for the model. Async Interrupt block and Task Sync block generated code is for the example RTOS (VxWorks). However, you can modify the blocks to generate code for another run-time environment.

1. Create a temporary folder for the build and inspection process.

2. Build the model.

```
Warning:  Simulink Coder: The tornado.tlc target will be removed in a future release.

### Wrapping unrecognized make command (angle brackets added)
###    <make>
### in default batch file
### Successfully updated the model reference RTW target for model: rtwdemo_async_mdlref
### Starting build procedure for model: rtwdemo_async_mdlreftop
Warning:  Simulink Coder: The tornado.tlc target will be removed in a future release.

### Wrapping unrecognized make command (angle brackets added)
###    <make>
### in default batch file
### Successful completion of code generation for model: rtwdemo_async_mdlreftop
```

Review Initialization Code

Open the generated source file `rtwdemo_async_mdlreftop.c`. The initialization code connects and enables ISR `isr_num1_vec192` for interrupt 1 and ISR `isr_num2_vec193` for interrupt 2.

```
static void rtwdemo_async_mdlreftop_initialize(void)
{
  /* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
  /* Connect and enable ISR function: isr_num1_vec192 */
  if (intConnect(INUM_TO_IVEC(192), isr_num1_vec192, 0) != OK) {
    printf("intConnect failed for ISR 1.\n");
  }

  sysIntEnable(1);

  /* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
  /* Connect and enable ISR function: isr_num2_vec193 */
  if (intConnect(INUM_TO_IVEC(193), isr_num2_vec193, 0) != OK) {
    printf("intConnect failed for ISR 2.\n");
  }

  sysIntEnable(2);

  /* SystemInitialize for ModelReference: '<Root>/Model' */
  rtwdemo_async_mdlrefbot_Init(&rtwdemo_async_mdlreftop_Y.Out1);

  /* Enable for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */
```

```
  /* Enable for ModelReference: '<Root>/Model' incorporates:
   *  Enable for Inport: '<Root>/In1_60hz'
   *  Enable for Inport: '<Root>/In2_60_hz'
   *  Enable for Inport: '<Root>/In3_60hz'
   */
  rtwdemo_async_mdlrefbot_Interrupt1_Enable();
  rtwdemo_async_mdlrefbot_Interrupt2_Enable();

  /* End of Enable for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */
}
```

Review ISR Code

In the generated source file `rtwdemo_async_mdlreftop.c`, review the code for ISRs `isr_num1_vec192` and `isr_num2_vec293`. Each ISR:

- Disables interrupts.
- Saves floating-point context.
- Calls the code generated for the subsystem connected to the referenced model Inport block that receives the interrupt.
- Restores floating-point context.
- Reenables interrupts.

```
void isr_num1_vec192(void)
{
  int_T lock;
  FP_CONTEXT context;

  /* disable interrupts (system is configured as non-preemptive) */
  lock = intLock();

  /* save floating point context */
  fppSave(&context);

  /* Call the system: '<Root>/Model' */
  {
    /* S-Function (vxinterrupt1): '<Root>/Async Interrupt' */

    /* ModelReference: '<Root>/Model' incorporates:
     *  Inport: '<Root>/In1_60hz'
     *  Inport: '<Root>/In2_60_hz'
```

```
     *  Inport: '<Root>/In3_60hz'
     */
    rtwdemo_async_mdlrefbot_Interrupt1(&rtwdemo_async_mdlreftop_Y.Out1);

    /* End of Outputs for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */
  }

  /* restore floating point context */
  fppRestore(&context);

  /* re-enable interrupts */
  intUnlock(lock);
}

/* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
void isr_num2_vec193(void)
{
  FP_CONTEXT context;

  /* save floating point context */
  fppSave(&context);

  /* Call the system: '<Root>/Model' */
  {
    /* S-Function (vxinterrupt1): '<Root>/Async Interrupt' */

    /* ModelReference: '<Root>/Model' incorporates:
     *  Inport: '<Root>/In1_60hz'
     *  Inport: '<Root>/In2_60_hz'
     *  Inport: '<Root>/In3_60hz'
     */
    rtwdemo_async_mdlrefbot_Interrupt2();

    /* End of Outputs for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */
  }

  /* restore floating point context */
  fppRestore(&context);
}
```

Review Task Termination Code

The Task Sync block generates the following termination code.

```
static void rtwdemo_async_mdlreftop_terminate(void)
{
  /* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
  /* Disable interrupt for ISR system: isr_num1_vec192 */
  sysIntDisable(1);

  /* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
  /* Disable interrupt for ISR system: isr_num2_vec193 */
  sysIntDisable(2);
}
```

Related Information

- Async Interrupt (Simulink Coder)
- Asynchronous Task Specification (Simulink Coder)
- “Generate Interrupt Service Routines” (Simulink Coder)
- “Timers in Asynchronous Tasks” (Simulink Coder)
- “Create a Customized Asynchronous Library” (Simulink Coder)
- “Import Asynchronous Event Data for Simulation” (Simulink Coder)
- “Load Data to Root-Level Input Ports” (Simulink)
- “Asynchronous Events” (Simulink Coder)
- “Rate Transitions and Asynchronous Blocks” (Simulink Coder)
- “Asynchronous Support Limitations” (Simulink Coder)

More About

- “Generate Interrupt Service Routines” (Simulink Coder)
- “Spawn and Synchronize Execution of RTOS Task” (Simulink Coder)
- “Timers in Asynchronous Tasks” (Simulink Coder)
- “Import Asynchronous Event Data for Simulation” (Simulink Coder)
- “Rate Transitions and Asynchronous Blocks” (Simulink Coder)

Rate Transitions and Asynchronous Blocks

Because an asynchronous function call subsystem can preempt or be preempted by other model code, an inconsistency arises when more than one signal element is connected to an asynchronous block. The issue is that signals passed to and from the function call subsystem can be in the process of being written to or read from when the preemption occurs. Thus, some old and some new data is used. This situation can also occur with scalar signals in some cases. For example, if a signal is a double (8 bytes), the read or write operation might require two machine instructions. The following sections describe these issues.

In this section...

Note: The operating system integration techniques that are demonstrated in this section use one or more blocks the blocks in the `vxlib1` (Simulink Coder) library. These blocks provide starting point examples to help you develop custom blocks for your target environment.

About Rate Transitions and Asynchronous Blocks

The Simulink Rate Transition block is designed to deal with preemption problems that occur in data transfer between blocks running at different rates. These issues are discussed in "Time-Based Scheduling and Code Generation" (Simulink Coder).

You can handle rate transition issues automatically by selecting the **Automatically handle data transfers between tasks** option on the **Solver** pane of the Configuration Parameters dialog box. This saves you from having to manually insert Rate Transition blocks to avoid invalid rate transitions, including invalid *asynchronous-to-periodic* and *asynchronous-to-asynchronous* rate transitions, in multirate models. For asynchronous tasks, the Simulink engine configures inserted blocks for data integrity but not determinism during data transfers.

For asynchronous rate transitions, the Rate Transition block provides data integrity, but cannot provide determinism. Therefore, when you insert Rate Transition blocks

explicitly, you must clear the **Ensure data determinism** check box in the Block Parameters dialog box.

When you insert a Rate Transition block between two blocks to maintain data integrity and priorities are assigned to the tasks associated with the blocks, the code generator assumes that the higher priority task can preempt the lower priority task and the lower priority task cannot preempt the higher priority task. If the priority associated with task for either block is not assigned or the priorities of the tasks for both blocks are the same, the code generator assumes that either task can preempt the other task.

Priorities of periodic tasks are assigned by the Simulink engine, in accordance with the options specified in the **Solver options** section of the **Solver** pane of the Configuration Parameters dialog box. When the **Periodic sample time constraint** option field of **Solver options** is set to `Unconstrained`, the model base rate priority is set to `40`. Priorities for subrates then increment or decrement by `1` from the base rate priority, depending on the setting of the **Higher priority value indicates higher task priority option**.

You can assign priorities manually by using the **Periodic sample time properties** field. The Simulink engine does not assign a priority to asynchronous blocks. For example, the priority of a function call subsystem that connects back to an Async Interrupt block is assigned by the Async Interrupt block.

The **Simulink task priority** field of the Async Interrupt block specifies a priority level (required) for every interrupt number entered in the **VME interrupt number(s)** field. The priority array sets the priorities of the subsystems connected to each interrupt.

For the Task Sync block, if the example RTOS (VxWorks) is the target, the **Higher priority value indicates higher task priority** option should be deselected. The **Simulink task priority** field specifies the block priority relative to connected blocks (in addition to assigning an RTOS priority to the generated task code).

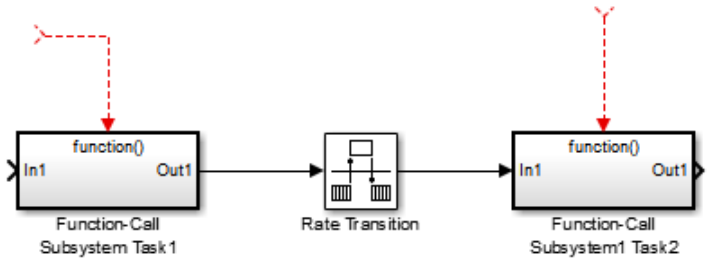
The `vxlib1` library provides two types of rate transition blocks as a convenience. These are simply preconfigured instances of the built-in Simulink Rate Transition block:

- Protected Rate Transition block: Rate Transition block that is configured with the **Ensure data integrity during data transfers** on and **Ensure deterministic data transfer** off.
- Unprotected Rate Transition block: Rate Transition block that is configured with the **Ensure data integrity during data transfers** option off.

Handle Rate Transitions for Asynchronous Tasks

For rate transitions that involve asynchronous tasks, you can maintain data integrity. However, you cannot achieve determinism. You have the option of using the Rate Transition block or target-specific rate transition blocks.

Consider the following model, which includes a Rate Transition block.



You can use the Rate Transition block in either of the following modes:

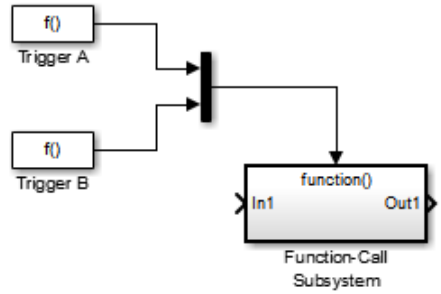
- Maintain data integrity, no determinism
- Unprotected

Alternatively, you can use target-specific rate transition blocks. The following blocks are available for the example RTOS (VxWorks):

- Protected Rate Transition block (reader)
- Protected Rate Transition block (writer)
- Unprotected Rate Transition block

Handle Multiple Asynchronous Interrupts

Consider the following model, in which two functions trigger the same subsystem.



The two tasks must have equal priorities. When priorities are the same, the outcome depends on whether they are firing periodically or asynchronously, and also on a diagnostic setting. The following table and notes describe these outcomes:

Supported Sample Time and Priority for Function Call Subsystem with Multiple Triggers

	Async Priority = 1	Async Priority = 2	Async Priority Unspecified	Periodic Priority = 1	Periodic Priority = 2
Async Priority = 1	Supported (1)				
Async Priority = 2		Supported (1)			
Async Priority Unspecified			Supported (2)		
Periodic Priority = 1				Supported	
Periodic Priority = 2					Supported

1. Control these outcomes using the **Tasks with equal priority** option in the **Diagnostics** pane of the Configuration Parameters dialog box; set this diagnostic to `none` if tasks of equal priority cannot preempt each other in the target system.
2. For this case, the following warning message is issued unconditionally:

```
The function call subsystem <name> has multiple asynchronous
triggers that do not specify priority. Data integrity will
not be maintained if these triggers can preempt one another.
```

Empty cells in the above table represent multiple triggers with differing priorities, which are unsupported.

The code generator provides absolute time management for a function call subsystem connected to multiple interrupts in the case where timer settings for `TriggerA` and `TriggerB` (time source, resolution) are the same.

Assume that all of the following conditions are true for the model shown above:

- A function call subsystem is triggered by two asynchronous triggers (TriggerA and TriggerB) having identical priority settings.
- Each trigger sets the source of time and timer attributes by calling the functions `ssSetTimeSource` and `ssSetAsyncTimerAttributes`.
- The triggered subsystem contains a block that needs elapsed or absolute time (for example, a Discrete Time Integrator).

The asynchronous function call subsystem has one global variable, `clockTick#` (where `#` is the task ID associated with the subsystem). This variable stores absolute time for the asynchronous task. There are two ways timing can be handled:

- If the time source is set to `SS_TIMESOURCE_BASERATE`, the code generator produces timer code in the function call subsystem, updating the clock tick variable from the base rate clock tick. Data integrity is maintained if the same priority is assigned to TriggerA and TriggerB.
- If the time source is `SS_TIMESOURCE_SELF`, generated code for both TriggerA and TriggerB updates the same clock tick variable from the hardware clock.

The word size of the clock tick variable can be set directly or be established according to the **Application lifespan (days)** (Simulink) setting and the timer resolution set by the TriggerA and TriggerB S-functions (which must be the same). See “Timers in Asynchronous Tasks” on page 17-44 and “Control Memory Allocation for Time Counters” on page 53-11 for more information.

More About

- “Time-Based Scheduling and Code Generation” (Simulink Coder)
- “Asynchronous Support Limitations” (Simulink Coder)

Timers in Asynchronous Tasks

An ISR can set a source for absolute time. This is done with the function `ssSetTimeSource`. The function `ssSetTimeSource` cannot be called before `ssSetOutputPortWidth` is called. If this occurs, the program will come to a halt and generate an error message. `ssSetTimeSource` has the following three options:

- `SS_TIMESOURCE_SELF`: Each generated ISR maintains its own absolute time counter, which is distinct from a periodic base rate or subrate counters in the system. The counter value and the timer resolution value (specified in the **Timer resolution (seconds)** parameter of the Async Interrupt block) are used by downstream blocks to determine absolute time values required by block computations.
- `SS_TIMESOURCE_CALLER`: The ISR reads time from a counter maintained by its caller. Time resolution is thus the same as its caller's resolution.
- `SS_TIMESOURCE_BASERATE`: The ISR can read absolute time from the model's periodic base rate. Time resolution is thus the same as its base rate resolution.

Note: The operating system integration techniques that are demonstrated in this section use one or more blocks the blocks in the `vxlib1` (Simulink Coder) library. These blocks provide starting point examples to help you develop custom blocks for your target environment.

By default, the counter is implemented as a 32-bit unsigned integer member of the `Timing` substructure of the real-time model structure. For a target that supports the `rtModel` data structure, when the time data type is not set by using `ssSetAsyncTimeDataType`, the counter word size is determined by the **Application lifespan (days)** (Simulink) model parameter. As an example (from ERT target code),

```
/* Real-time Model Data Structure */
struct _RT_MODEL_elapseTime_exp_Tag {
   const char *errorStatus;

  /*
   * Timing:
   * The following substructure contains information regarding
   * the timing information for the model.
   */
  struct {
    uint32_T clockTick1;
```

```
    uint32_T clockTick2;
  } Timing;
};
```

The example omits unused fields in the `Timing` data structure (a feature of ERT target code not found in GRT). For a target that supports the `rtModel` data structure, the counter word size is determined by the **Application lifespan (days)** (Simulink) model parameter.

By default, the `vxlib1` library blocks for the example RTOS (VxWorks) set the timer source to `SS_TIMESOURCE_SELF` and update their counters by using the system call `tickGet`. `tickGet` returns a timer value maintained by the RTOS kernel. The maximum word size for the timer is `UINT32`. The following example shows a generated call to `tickGet`.

```
/* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
void isr_num2_vec193(void)
{

 /* Use tickGet() as a portable tick counter example. A much
    higher resolution can be achieved with a hardware counter */
 rtM->Timing.clockTick2 = tickGet();
. . .
```

The `tickGet` call is supplied only as an example. It can (and in many instances should) be replaced by a timing source that has better resolution. If you are implementing a custom asynchronous block for an RTOS other than the example RTOS (VxWorks), you should either generate an equivalent call to the target RTOS, or generate code to read a timer register on the target hardware.

The default **Timer resolution (seconds)** parameter of your Async Interrupt block implementation should be changed to match the resolution of your target's timing source.

The counter is updated at interrupt level. Its value represents the tick value of the timing source at the most recent execution of the ISR. The rate of this timing source is unrelated to sample rates in the model. In fact, typically it is faster than the model's base rate. Select the timer source and set its rate and resolution based on the expected rate of interrupts to be serviced by the Async Interrupt block.

For an example of timer code generation, see “Async Interrupt Block Implementation” on page 17-48.

Related Examples

- “Generate Interrupt Service Routines” on page 17-6
- “Spawn and Synchronize Execution of RTOS Task” on page 17-15
- “Timers in Asynchronous Tasks” on page 17-44
- “Create a Customized Asynchronous Library” on page 17-47
- “Import Asynchronous Event Data for Simulation” on page 17-56

More About

- “Absolute and Elapsed Time Computation” (Simulink Coder)
- “Time-Based Scheduling and Code Generation” (Simulink Coder)
- “Asynchronous Events” (Simulink Coder)
- “Asynchronous Support Limitations” (Simulink Coder)

Create a Customized Asynchronous Library

This topic describes how to implement asynchronous blocks for use with your target RTOS, using the Async Interrupt and Task Sync blocks as a starting point. Rate Transition blocks are target-independent, so you do not need to develop customized rate transition blocks. The following sections provide implementation details.

In this section...

Note: The operating system integration techniques that are demonstrated in this section use one or more blocks the blocks in the `vxlib1` (Simulink Coder) library. These blocks provide starting point examples to help you develop custom blocks for your target environment.

About Implementing Asynchronous Blocks

You can customize the asynchronous library blocks by modifying the block implementation. These files are

- The block's underlying S-function MEX-file
- The TLC files that control code generation of the block

In addition, you need to modify the block masks to remove references specific to the example RTOS (VxWorks) and to incorporate parameters required by your target RTOS.

Custom block implementation is an advanced topic, requiring familiarity with the Simulink MEX S-function format and API, and with the Target Language Compiler (TLC). These topics are covered in the following documents:

- Simulink topics "What Is an S-Function?" (Simulink), "Use S-Functions in Models" (Simulink), "How S-Functions Work" (Simulink), and "Implementing S-Functions" (Simulink) describe MEX S-functions and the S-function API in general.

- The “Inlining S-Functions” (Simulink Coder), “Inline C MEX S-Functions” (Simulink Coder), and “S-Functions and Code Generation” (Simulink Coder) describe how to create a TLC block implementation for use in code generation.

The following sections discuss the C/C++ and TLC implementations of the asynchronous library blocks, including required `SimStruct` macros and functions in the TLC asynchronous support library (`asynclib.tlc`).

Async Interrupt Block Implementation

The source files for the Async Interrupt block are located in *`matlabroot`*`/rtw/c/tornado/devices` (open):

- `vxinterrupt1.c`: C MEX-file source code, for use in configuration and simulation
- `vxinterrupt1.tlc`: TLC implementation, for use in code generation
- `asynclib.tlc`: library of TLC support functions, called by the TLC implementation of the block. The library calls are summarized in “asynclib.tlc Support Library” on page 17-53.

C MEX Block Implementation

Most of the code in `vxinterrupt1.c` performs ordinary functions that are not related to asynchronous support (for example, obtaining and validating parameters from the block mask, marking parameters nontunable, and passing parameter data to the *`model`*`.rtw` file).

The `mdlInitializeSizes` function uses special `SimStruct` macros and `SS_OPTIONS` settings that are required for asynchronous blocks, as described below.

Note that the following macros cannot be called before `ssSetOutputPortWidth` is called:

- `ssSetTimeSource`
- `ssSetAsyncTimerAttributes`
- `ssSetAsyncTimerResolutionEl`
- `ssSetAsyncTimerDataType`
- `ssSetAsyncTimerDataTypeEl`
- `ssSetAsyncTaskPriorities`
- `ssSetAsyncTaskPrioritiesEl`

If one of the above macros is called before `ssSetOutputPortWidth`, the following error message appears:

```
SL_SfcnMustSpecifyPortWidthBfCallSomeMacro {
S-function '%s' in '%<BLOCKFULLPATH>'
must set output port %d width using
ssSetOutputPortWidth before calling macro %s
 }
```

ssSetAsyncTimerAttributes

`ssSetAsyncTimerAttributes` declares that the block requires a timer, and sets the resolution of the timer as specified in the **Timer resolution (seconds)** parameter.

The function prototype is

```
ssSetAsyncTimerAttributes(SimStruct *S, double res)
```

where

- S is a Simstruct pointer.
- `res` is the **Timer resolution (seconds)** parameter value.

The following code excerpt shows the call to ssSetAsyncTimerAttributes.

```
/* Setup Async Timer attributes */
ssSetAsyncTimerAttributes(S,mxGetPr(TICK_RES)[0]);
```

ssSetAsyncTaskPriorities

`ssSetAsyncTaskPriorities` sets the Simulink task priority for blocks executing at each interrupt level, as specified in the block's **Simulink task priority** field.

The function prototype is

```
ssSetAsyncTaskPriorities(SimStruct *S, int numISRs,
                         int *priorityArray)
```

where

- `S` is a `SimStruct` pointer.
- `numISRs` is the number of interrupts specified in the **VME interrupt number(s)** parameter.
- `priorityarray` is an integer array containing the interrupt numbers specified in the **VME interrupt number(s)** parameter.

The following code excerpt shows the call to `ssSetAsyncTaskPriorities`:

```
/* Setup Async Task Priorities */
    priorityArray = malloc(numISRs*sizeof(int_T));
    for (i=0; i<numISRs; i++) {
        priorityArray[i] = (int_T)(mxGetPr(ISR_PRIORITIES)[i]);
    }
    ssSetAsyncTaskPriorities(S, numISRs, priorityArray);
    free(priorityArray);
    priorityArray = NULL;
}
```

SS_OPTION Settings

The code excerpt below shows the `SS_OPTION` settings for `vxinterrupt1.c`. `SS_OPTION_ASYNCHRONOUS_INTERRUPT` should be used when a function call subsystem is attached to an interrupt. For more information, see the documentation for `SS_OPTION` and `SS_OPTION_ASYNCHRONOUS` in *`matlabroot`*`/simulink/include/simstruc.h`.

```
ssSetOptions( S, (SS_OPTION_EXCEPTION_FREE_CODE |
             SS_OPTION_DISALLOW_CONSTANT_SAMPLE_TIME |
             SS_OPTION_ASYNCHRONOUS_INTERRUPT |
```

TLC Implementation

This section discusses each function of `vxinterrupt1.tlc`, with an emphasis on target-specific features that you will need to change to generate code for your target RTOS.

Generate #include Directives

`vxinterrupt1.tlc` begins with the statement

```
%include "vxlib.tlc"
```

`vxlib.tlc` is a target-specific file that generates directives to include header files for the example RTOS (VxWorks). You should replace this with a file that generates includes for your target RTOS.

BlockInstanceSetup Function

For each connected output of the Async Interrupt block, `BlockInstanceSetup` defines a function name for the corresponding ISR in the generated code. The functions names are of the form

`isr_`*`num`*`_vec_`*`offset`*

where *num* is the ISR number defined in the **VME interrupt number(s)** block parameter, and *offset* is an interrupt table offset defined in the **VME interrupt vector offset(s)** block parameter.

In a custom implementation, this naming convention is optional.

The function names are cached for use by the `Outputs` function, which generates the actual ISR code.

Outputs Function

`Outputs` iterates over the connected outputs of the Async Interrupt block. An ISR is generated for each such output.

The ISR code is cached in the `"Functions"` section of the generated code. Before generating the ISR, `Outputs` does the following:

- Generates a call to the downstream block (cached in a temporary buffer).
- Determines whether the ISR should be locked or not (as specified in the **Preemption Flag(s)** block parameter).
- Determines whether the block connected to the Async Interrupt block is a Task Sync block. (This information is obtained by using the `asynclib` calls `LibGetFcnCallBlock` and `LibGetBlockAttribute`.) If so,
  - The preemption flag for the ISR must be set to `1`. An error results otherwise.
  - The RTOS (VxWorks) calls to save and restore floating-point context are generated, unless the user has configured the model for integer-only code generation.

When generating the ISR code, `Outputs` calls the `asynclib` function `LibNeedAsyncCounter` to determine whether a timer is required by the connected subsystem. If so, and if the time source is set to be `SS_TIMESOURCE_SELF` by `ssSetTimeSource`, `LibSetAsyncCounter` is called to generate an RTOS (VxWorks) `tickGet` function call and update the counter. In your implementation, you should generate either an equivalent call to the target RTOS, or generate code to read the a timer register on the target hardware.

Start Function

The `Start` function generates the required RTOS (VxWorks) calls (`int_connect` and `sysInt_Enable`) to connect and enable each ISR. You should replace this with calls to your target RTOS.

Terminate Function

The `Terminate` function generates the call `sysIntDisable` to disable each ISR. You should replace this with calls to your target RTOS.

Task Sync Block Implementation

The source files for the Task Sync block are located in *`matlabroot`*`/rtw/c/tornado/devices` (open). They are

- `vxtask1.cpp`: MEX-file source code, for use in configuration and simulation.
- `vxtask1.tlc`: TLC implementation, for use in code generation.
- `asynclib.tlc`: library of TLC support functions, called by the TLC implementation of the block. The library calls are summarized in "asynclib.tlc Support Library" on page 17-53.

C MEX Block Implementation

Like the Async Interrupt block, the Task Sync block sets up a timer, in this case with a fixed resolution. The priority of the task associated with the block is obtained from the **Simulink task priority** parameter. The `SS_OPTION` settings are the same as those used for the Async Interrupt block.

```
ssSetAsyncTimerAttributes(S, 0.01);

priority  = (int_T) (*(mxGetPr(PRIORITY)));
ssSetAsyncTaskPriorities(S,1,&priority);

ssSetOptions(S, (SS_OPTION_EXCEPTION_FREE_CODE |
                 SS_OPTION_ASYNCHRONOUS |
                 SS_OPTION_DISALLOW_CONSTANT_SAMPLE_TIME |
}
```

TLC Implementation

Generate #include Directives

`vxtask1.tlc` begins with the statement

```
%include "vxlib.tlc"
```

`vxlib.tlc` is a target-specific file that generates directives to include header files for the example RTOS (VxWorks). You should replace this with a file that generates includes for your target RTOS.

BlockInstanceSetup Function

The `BlockInstanceSetup` function derives the task name, block name, and other identifiers used later in code generation. It also checks for and warns about unconnected block conditions, and generates a storage declaration for a semaphore (`stopSem`) that is used in case of interrupt overflow conditions.

Start Function

The `Start` function generates the required RTOS (VxWorks) calls to define storage for the semaphore that is used in management of the task spawned by the Task Sync block. Depending on the value of the `CodeFormat` TLC variable of the target, either a static storage declaration or a dynamic memory allocation call is generated. This function also creates a semaphore (`semBCreate`) and spawns an RTOS task (`taskSpawn`). You should replace these with calls to your target RTOS.

Outputs Function

The `Outputs` function generates an example RTOS (VxWorks) task that waits for a semaphore. When it obtains the semaphore, it updates the block's tick timer and calls the downstream subsystem code, as described in "Spawn and Synchronize Execution of RTOS Task" on page 17-15. `Outputs` also generates code (called from interrupt level) that grants the semaphore.

Terminate Function

The `Terminate` function generates the example RTOS (VxWorks) call `taskDelete` to end execution of the task spawned by the block. You should replace this with calls to your target RTOS.

Note also that if the target RTOS has dynamically allocated memory associated with the task, the `Terminate` function should deallocate the memory.

asynclib.tlc Support Library

`asynclib.tlc` is a library of TLC functions that support the implementation of asynchronous blocks. Some functions are specifically designed for use in asynchronous blocks. For example, `LibSetAsyncCounter` generates a call to update a timer for an asynchronous block. Other functions are utilities that return information required by asynchronous blocks (for example, information about connected function call subsystems).

The following table summarizes the public calls in the library. For details, see the library source code and the `vxinterrupt1.tlc` and `vxtask1.tlc` files, which call the library functions.

Summary of asynclib.tlc Library Functions

Function	Description
`LibBlockExecuteFcnCall`	For use by inlined S-functions with function call outputs. Generates code to execute a function call subsystem.
`LibGetBlockAttribute`	Returns a field value from a block record.
`LibGetFcnCallBlock`	Given an S-Function block and call index, returns the block record for the downstream function call subsystem block.
`LibGetCallerClockTickCounter`	Provides access to the time counter of an upstream asynchronous task.
`LibGetCallerClockTickCounter-HighWord`	Provides access to the high word of the time counter of an upstream asynchronous task.
`LibManageAsyncCounter`	Determines whether an asynchronous task needs a counter and manages its own timer.
`LibNeedAsyncCounter`	If the calling block requires an asynchronous counter, returns `TLC_TRUE`, otherwise returns `TLC_FALSE`.
`LibSetAsyncClockTicks`	Returns code that sets `clockTick` counters that are to be maintained by the asynchronous task.
`LibSetAsyncCounter`	Generates code to set the tick value of the block's asynchronous counter.
`LibSetAsyncCounterHighWord`	Generates code to set the tick value of the high word of the block's asynchronous counter

More About

- “Asynchronous Events” (Simulink Coder)
- “Generate Interrupt Service Routines” (Simulink Coder)
- “Spawn and Synchronize Execution of RTOS Task” (Simulink Coder)
- “Pass Asynchronous Events in RTOS as Input To a Referenced Model” (Simulink Coder)
- “Timers in Asynchronous Tasks” (Simulink Coder)

- “Import Asynchronous Event Data for Simulation” (Simulink Coder)
- “Rate Transitions and Asynchronous Blocks” (Simulink Coder)
- “Asynchronous Support Limitations” (Simulink Coder)

Import Asynchronous Event Data for Simulation

Capabilities

You can import asynchronous event data into a function-call subsystem via an Inport block. For standalone fixed-step simulations, you can specify:

- The time points at which each asynchronous event occurs
- The number of asynchronous events at each time point

Input Data Format

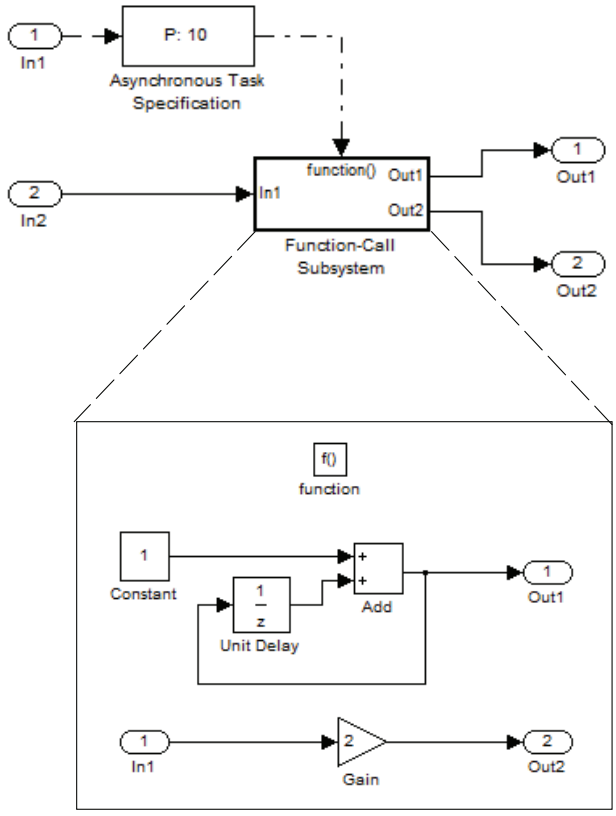
You can enter your asynchronous data at the MATLAB command line or on the **Data Import/Export** pane of the Configuration Parameters dialog box. In either case, a number of restrictions apply to the data format.

- The expression for the parameter **Data Import/Export > Input** must be a comma-separated list of tables.
- The table corresponding to the input port outputting asynchronous events must be a column vector containing time values for the asynchronous events.
  - The time vector of the asynchronous events must be of double data type and monotonically increasing.
  - All time data must be integer multiples of the model step size.
  - To specify multiple function calls at a given time step, you must repeat the time value accordingly. In other words, if you wish to specify three asynchronous events at $t = 1$ and two events at $t = 9$, then you must list `1` three times and `9` twice in your time vector. (`t = [1 1 1 9 9]'`)
- The table corresponding to normal data input port can be of any other supported format.

See “Load Data to Root-Level Input Ports” (Simulink) for more information.

Example

In this model, a function-call subsystem is used to track the total number of asynchronous events and to multiply a set of inputs by 2.



1. To input data via the Configuration Parameters dialog box,

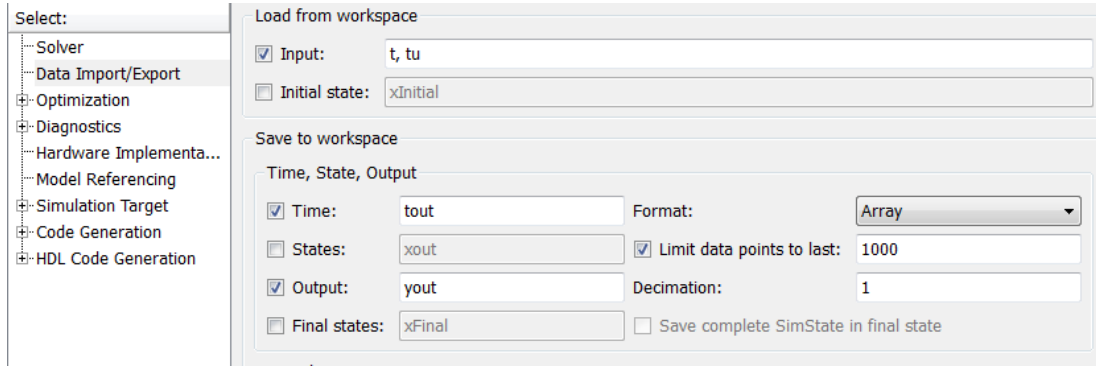
   a. Select **Simulation > Configuration Parameters > Data Import/Export**.

   b. Select the **Input** parameter.

   c. For this example, enter the following command in the MATLAB window:

   ```
   >> t = [1 1 5 9 9 9]', u = [[0:10]' [0:10]']
   ```

   Alternatively, you can enter the data as *t, tu* in the Data Import/Export pane:



Here, *t* is a column vector containing the times of asynchronous events for Inport block In1 while *tu* is a table of input values versus time for Inport block In2.

2. By default, the **Time** and **Output** options are selected and the output variables are named *tout* and *yout*.
3. Simulate the model.
4. Display the output by entering `[tout yout]` at the MATLAB command line and obtain:

```
ans =

     0     0    -1
     1     2     2
     2     2     2
     3     2     2
     4     2     2
     5     3    10
     6     3    10
     7     3    10
     8     3    10
     9     6    18
    10     6    18
```

Here the first column contains the simulation times.

The second column represents the output of Out1 — the total number of asynchronous events. Since the function-call subsystem is triggered twice at $t = 1$,

the output is 2. It is not called again until $t = 5$, and so does not increase to 3 until then. Finally, it is called three times at 9, so it increases to 6.

The third column contains the output of Out2 obtained by multiplying the input value at each asynchronous event time by 2. At any other time, the output is held at its previous value

More About

- "Asynchronous Events" (Simulink Coder)
- "Pass Asynchronous Events in RTOS as Input To a Referenced Model" (Simulink Coder)
- "Load Data to Root-Level Input Ports" (Simulink)
- "Rate Transitions and Asynchronous Blocks" (Simulink Coder)

Asynchronous Support Limitations

In this section...

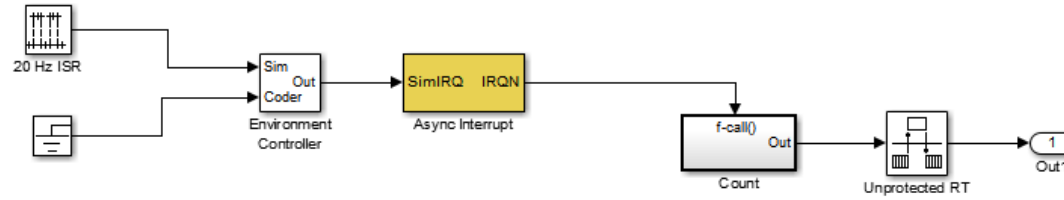
Asynchronous Task Priority

The Simulink product does not simulate asynchronous task behavior. Although you can specify a task priority for an asynchronous task represented in a model with the Task Sync block, the priority setting is for code generation purposes only and is not honored during simulation.

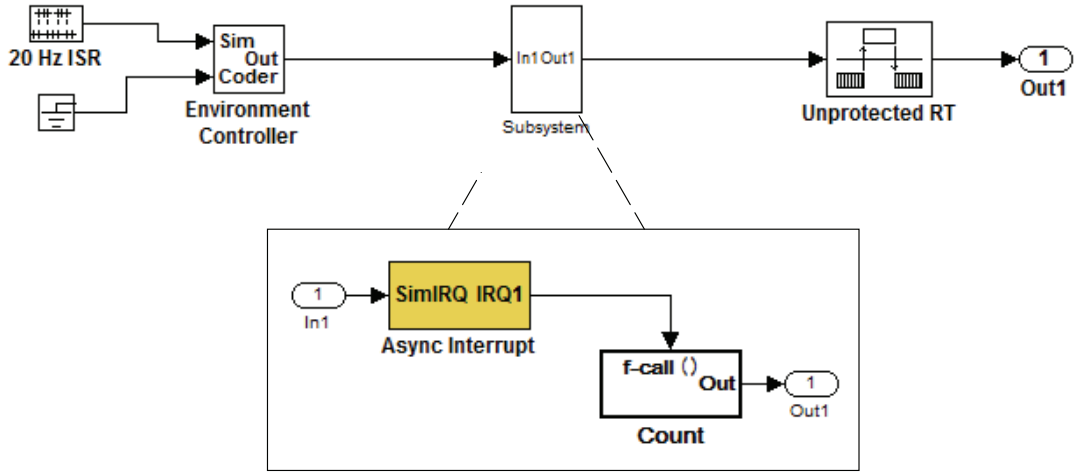
Convert an Asynchronous Subsystem into a Model Reference

You can use the Asynchronous Task Specification block to specify an asynchronous function-call input to a model reference. However, you must convert the Async Interrupt and Function-Call blocks into a subsystem and then convert the subsystem into a model reference.

Following is an example with step-by-step instructions for conversion.



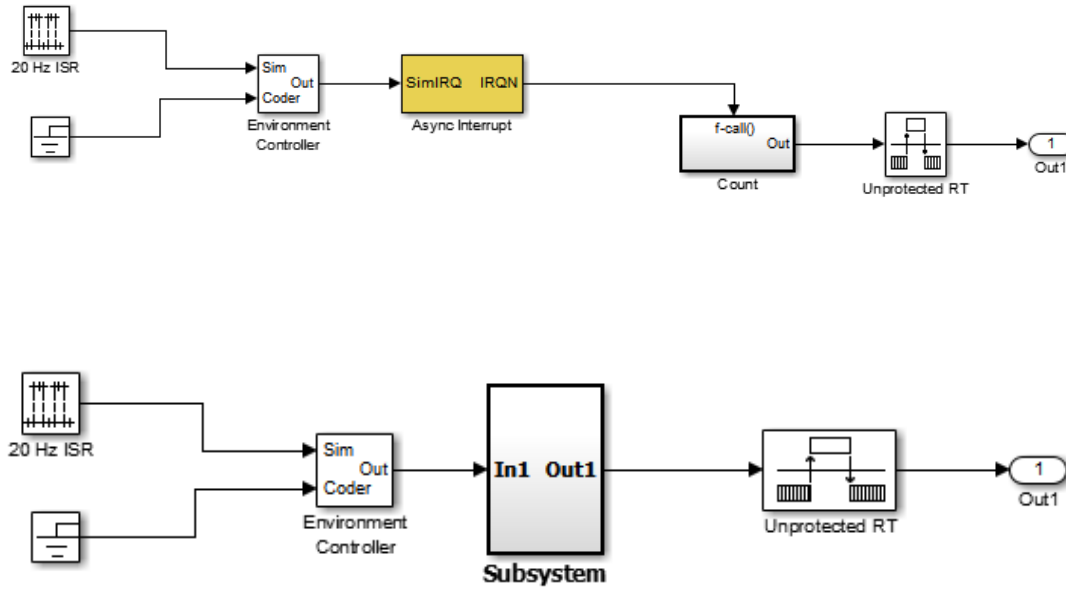
1 Convert the Async Interrupt and Count blocks into a subsystem. Select both blocks and right-click Count. From the menu, select **Subsystem & Model Reference > Create Subsystem from Selection**.



2 To prepare for converting the new subsystem to a Model block, set the following configuration parameters in the top model. Open the Configuration Parameters dialog box.

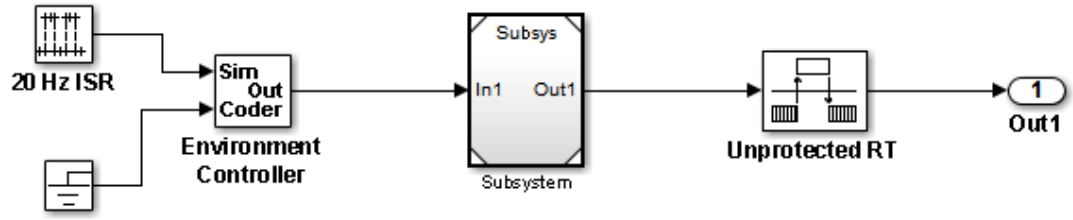
- Under Diagnostics, navigate to the Sample Time pane. Then set **Multitask rate transition** to `error` and **Multitask conditionally executed subsystem** to `error`.
- Under Diagnostics, navigate to the Connectivity pane. Set **Bus signal treated as vector**, and **Invalid function-call connection** to `error`. Also set **Context-dependent inputs** to `Enable All`.
- Under Diagnostics, navigate to the Data Validity pane and set the **Multitask data store** option to `error`.
- On the **All Parameters** tab, set **Underspecified initialization detection** to `Simplified`.
- If your model is large or complex, run the Model Advisor checks in the folder "Migrating to Simplified Initialization Mode Overview" (Simulink) and make the suggested changes.

3 Convert the subsystem to an atomic subsystem. Select **Edit > Subsystem Parameters > Treat as atomic unit**.

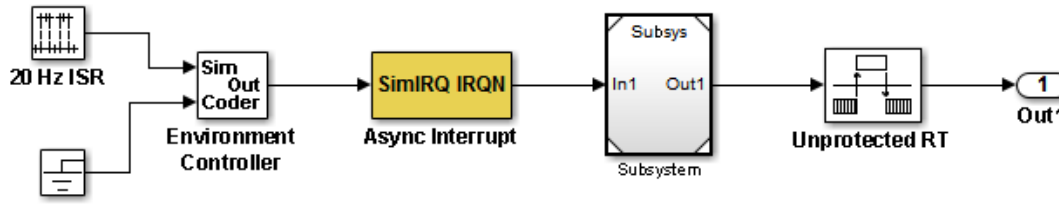


4 Convert the subsystem to a Model block. Right-click the subsystem and select **Subsystem & Model Reference > Convert Subsystem to > Referenced Model**. A window opens with a model reference block inside of it.

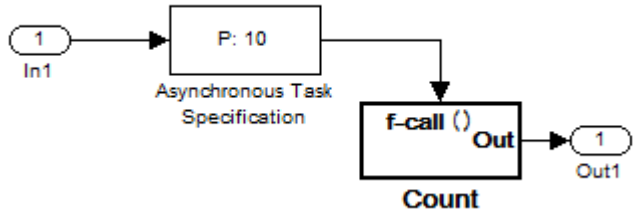
5 Replace the subsystem in the top model with the new model reference block.



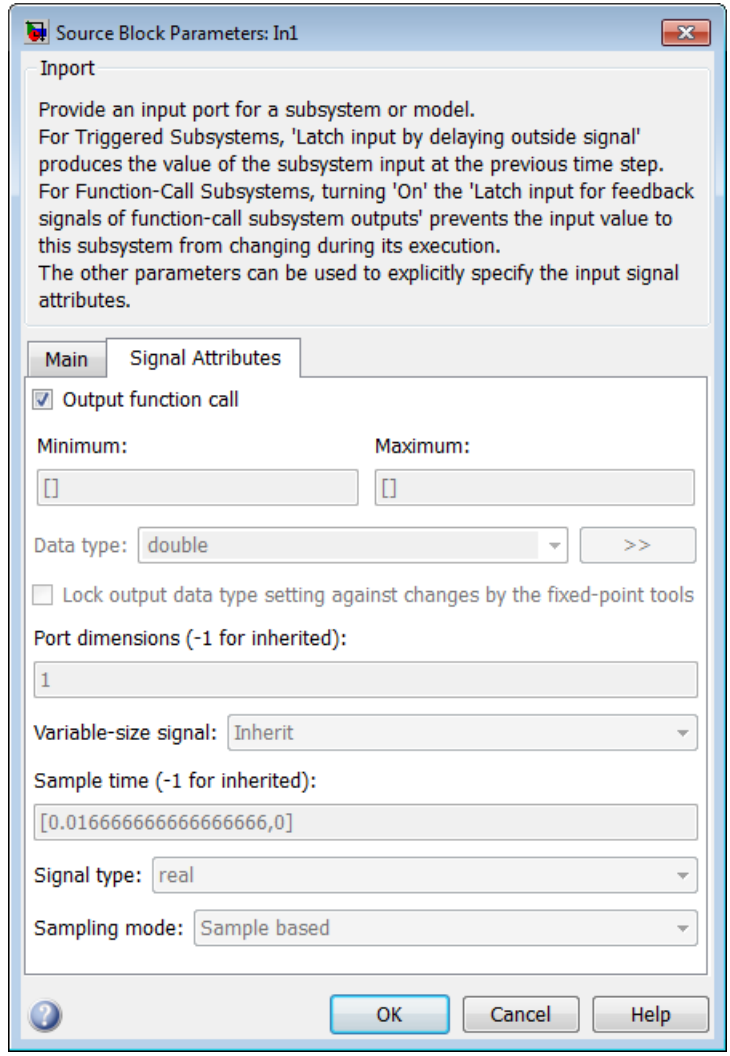
6 Move the Async Interrupt block from the model reference to the top model, before the model reference block.



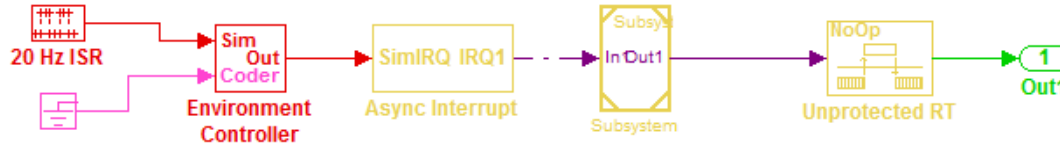
7 Insert an Asynchronous Task Specification block in the model reference. Set the priority of the Asynchronous Task Specification block. (For more information on setting the priority, see Asynchronous Task Specification (Simulink Coder).)



8 In the model reference, double-click the input port to open its Source Block Parameters dialog box. Click the **Signal Attributes** tab and select the **Output function call** option. Click **OK**.



9 Save your model and then perform **Simulation > Update Diagram** to verify your settings.



More About

- "Asynchronous Events" (Simulink Coder)

18

Scheduling Considerations in Embedded Coder

Use Discrete and Continuous Time

In this section...

Support for Discrete and Continuous Time Blocks

The ERT target supports code generation for discrete and continuous time blocks. If the **Support: continuous time** option is selected on the **Code Generation > Interface** pane, you can use these 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 Designer block libraries, including whether or not they are recommended for use in production code generation. To view this table, execute the following command and see the "Code Generation Support" column of the table that appears:

```
showblockdatatypetable
```

Support for Continuous Solvers

The ERT target supports continuous solvers. In the **Solver** options dialog, you can select an 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 "Customize System Target Files" (Simulink Coder).

Support for Stop Time

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

- **External mode** is selected on the **Code Generation > Interface** pane

- **MAT-file logging** is selected on the **All Parameters** tab
- **Classic call interface** is selected on the **All Parameters** tab

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.

More About

- "Time-Based Scheduling and Code Generation" on page 16-2
- "Configure Time-Based Scheduling" on page 16-34
- "Sample Times in Subsystems" (Simulink)
- "Sample Times in Systems" (Simulink)

Optimize Multirate Multitasking Execution for RTOS Run-Time Environments

Using the `rtmStepTask` macro, run-time environments that employ task management mechanisms of an real-time operating system (RTOS)—for example, VxWorks—can improve performance of generated code by eliminating redundant scheduling calls during the execution of tasks in a multirate, multitasking model, The following sections describe implementation details.

Use rtmStepTask

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

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

The arguments are:

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

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

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

Schedule Code for Real-time Model without an RTOS

To understand the optimization that is available for an RTOS target, consider how the ERT target schedules tasks for bareboard targets (where RTOS is not present). The ERT target maintains *scheduling counters* and *event flags* for each subrate task. The scheduling counters are implemented within the real-time model (rtM) data structure as arrays, indexed on task identifier (`tid`).

The scheduling counters are updated by the base-rate task. The counters are clock rate dividers that count up the sample period associated with each subrate task. When a given subrate counter reaches a value that indicates it has a hit, the sample period for

that rate has elapsed and the counter is reset to zero. When this occurs, the subrate task must be scheduled for execution.

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

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

Schedule Code for Multirate Multitasking on an RTOS

The following task scheduling code, from `ertmainlib.tlc`, is designed for multirate multitasking operation on an example RTOS (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 RTOS `semGive` function is called, and the RTOS schedules the task to run.

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

Suppress 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
```

More About

- “Time-Based Scheduling and Code Generation” on page 16-2
- “Modeling for Multitasking Execution” on page 16-12

Data, Function, and File Definition

Data Representation in Simulink Coder

Access Signal, State, and Parameter Data During Execution

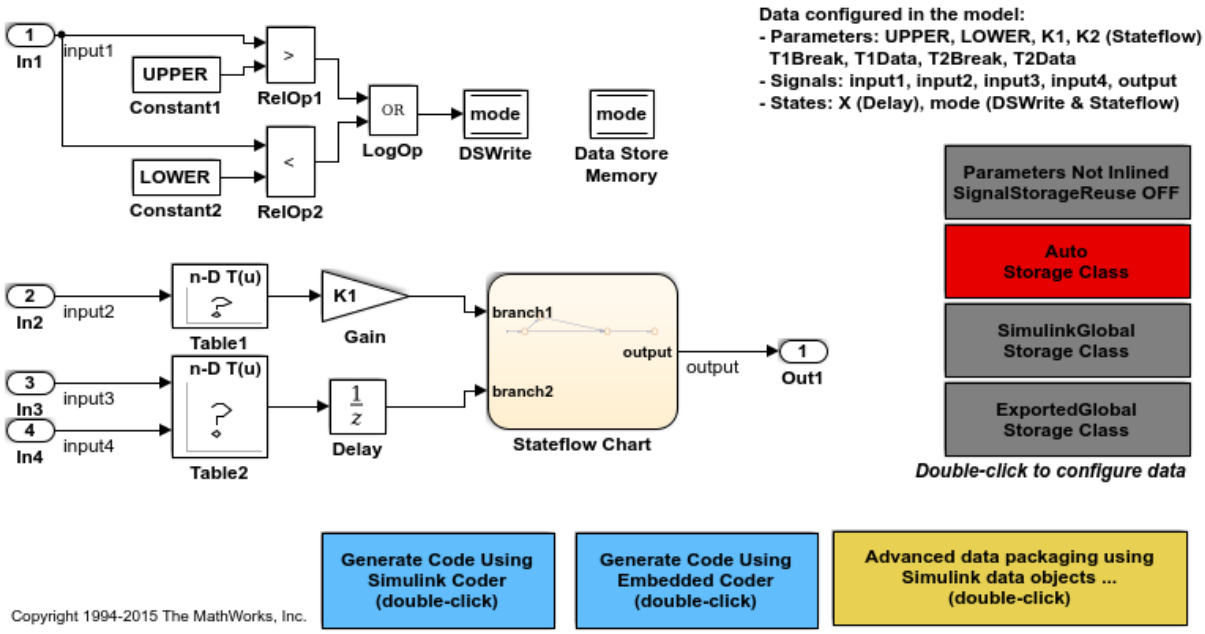
As you iteratively develop a model, you capture output signal and state data that model execution generates. You also tune parameter values during execution to observe the effect on the outputs. You can then base your design decisions upon analysis of these outputs. To access this signal, state, and parameter data in a rapid prototyping environment, you can configure the generated code to store the data in addressable memory.

By default, optimization settings make the generated code more efficient by eliminating unnecessary signal storage and inlining the numeric values of block parameters. To generate code that instead allocates addressable memory for this data, you can disable the optimizations or specify code generation settings for individual data items.

Explore Example Model

Open the example model rtwdemo_basicsc.

```
rtwdemo_basicsc
```



The model loads numeric MATLAB variables, such as `K1`, into the base workspace.

In the model, open the block dialog box for the Gain block labeled `Gain`. The block uses the variable `K1` to set the value of the **Gain** parameter.

Disable Optimizations

In the model, clear the model configuration parameter **Signal storage reuse**. When you clear this optimization and other optimizations such as **Eliminate superfluous local variables (expression folding)**, the generated code allocates memory for signal lines. Clearing **Signal storage reuse** disables most of the other optimizations.

```
set_param('rtwdemo_basicsc','OptimizeBlockIOStorage','off')
```

Set the optimization **Configuration Parameters > Optimization > Signals and Parameters > Default parameter behavior** to `Tunable`. When set to `Tunable`, this configuration parameter causes the generated code to allocate memory for block parameters and workspace variables.

```
set_param('rtwdemo_basicsc','DefaultParameterBehavior','Tunable')
```

Generate code from the model.

```
rtwbuild('rtwdemo_basicsc')
```

```
### Starting build procedure for model: rtwdemo_basicsc
### Successful completion of build procedure for model: rtwdemo_basicsc
```

In the code generation report, view the file `rtwdemo_basicsc.h`. This header file defines a structure type that contains signal data. The structure contains fields that each represent a signal line in the model. For example, the output signal of the Gain block labeled `Gain` appears as the field `Gain`.

```
file = fullfile('rtwdemo_basicsc_grt_rtw','rtwdemo_basicsc.h');
rtwdemodbtype(file,'/* Block signals (auto storage) */',...
    'B_rtwdemo_basicsc_T;',1,1)
```

```
/* Block signals (auto storage) */
typedef struct {
  real32_T Table1;                     /* '<Root>/Table1' */
  real32_T Gain;                       /* '<Root>/Gain' */
  real32_T Delay;                      /* '<Root>/Delay' */
```

```
  real32_T Table2;                     /* '<Root>/Table2' */
  boolean_T RelOp1;                    /* '<Root>/RelOp1' */
  boolean_T RelOp2;                    /* '<Root>/RelOp2' */
  boolean_T LogOp;                     /* '<Root>/LogOp' */
} B_rtwdemo_basicsc_T;
```

The file defines a structure type that contains block parameter data. The MATLAB variable `K1` appears as a field of the structure. The other fields of the structure represent other block parameters and workspace variables from the model, including initial conditions for signals.

```
rtwdemodbtype(file,'/* Parameters (auto storage) */',...
    '/* Real-time Model Data Structure */',1,0)
```

```
/* Parameters (auto storage) */
struct P_rtwdemo_basicsc_T_ {
  real_T K2;                           /* Variable: K2
                                        * Referenced by: '<Root>/Stateflow Chart'
                                        */
  real32_T LOWER;                      /* Variable: LOWER
                                        * Referenced by: '<Root>/Constant2'
                                        */
  real32_T T1Break[11];                /* Variable: T1Break
                                        * Referenced by: '<Root>/Table1'
                                        */
  real32_T T1Data[11];                 /* Variable: T1Data
                                        * Referenced by: '<Root>/Table1'
                                        */
  real32_T T2Break[3];                 /* Variable: T2Break
                                        * Referenced by: '<Root>/Table2'
                                        */
  real32_T T2Data[9];                  /* Variable: T2Data
                                        * Referenced by: '<Root>/Table2'
                                        */
  real32_T UPPER;                      /* Variable: UPPER
                                        * Referenced by: '<Root>/Constant1'
                                        */
  int8_T K1;                           /* Variable: K1
                                        * Referenced by: '<Root>/Gain'
                                        */
  real32_T Delay_InitialCondition;     /* Computed Parameter: Delay_InitialCondition
                                        * Referenced by: '<Root>/Delay'
                                        */
  uint32_T Table2_maxIndex[2];         /* Computed Parameter: Table2_maxIndex
```

```
                                        * Referenced by: '<Root>/Table2'
                                        */
  boolean_T DataStoreMemory_InitialValue;/* Computed Parameter: DataStoreMemory_Initial
                                          * Referenced by: '<Root>/Data Store Memory'
                                          */
};
```

View the file `rtwdemo_basicsc_data.c`. This source file allocates global memory for a parameter structure and initializes the field values based on the parameter values in the model.

View the source file `rtwdemo_basicsc.c`. The code allocates global memory for a structure variable that contains signal data.

```
file = fullfile('rtwdemo_basicsc_grt_rtw','rtwdemo_basicsc.c');
rtwdemodbtype(file,'/* Block signals (auto storage) */',...
    'B_rtwdemo_basicsc_T rtwdemo_basicsc_B;',1,1)
```

```
/* Block signals (auto storage) */
B_rtwdemo_basicsc_T rtwdemo_basicsc_B;
```

The code algorithm in the model `step` function calculates the signal values. It then assigns these values to the fields of the signal structure. To perform the calculations, the algorithm uses the parameter values from the fields of the parameter structure.

Exclude Data Items from Optimizations

When you want to select code generation optimizations such as **Signal storage reuse**, you can preserve individual data items from the optimizations. The generated code then allocates addressable memory for the items.

Select the optimizations that you previously cleared.

```
set_param('rtwdemo_basicsc','OptimizeBlockIOStorage','on')
set_param('rtwdemo_basicsc','LocalBlockOutputs','on')
set_param('rtwdemo_basicsc','DefaultParameterBehavior','Inlined')
```

Right-click the output of the Gain block labeled `Gain` and select **Properties**. In the Signal Properties dialog box, select **Test point**.

```
portHandle = get_param('rtwdemo_basicsc/Gain','PortHandles');
portHandle = portHandle.Outport;
set_param(portHandle,'TestPoint','on')
```

Convert the MATLAB variable `K1` to a `Simulink.Parameter` object. With parameter objects, you can create addressable parameters to tune during execution of the generated code.

```
K1 = Simulink.Parameter(K1);
```

Apply a storage class other than `Auto` to the parameter object `K1`. For example, use the storage class `SimulinkGlobal` to represent the parameter object as a field of the global parameter structure.

```
K1.StorageClass = 'SimulinkGlobal';
```

Generate code from the model.

```
rtwbuild('rtwdemo_basicsc')
```

```
### Starting build procedure for model: rtwdemo_basicsc
### Successful completion of build procedure for model: rtwdemo_basicsc
```

In the code generation report, view the file `rtwdemo_basicsc.h`. The structure that contains signal data now defines only one field, `Gain`, which represents the test-pointed output of the Gain block.

```
file = fullfile('rtwdemo_basicsc_grt_rtw','rtwdemo_basicsc.h');
rtwdemodbtype(file,'/* Block signals (auto storage) */',...
    'B_rtwdemo_basicsc_T;',1,1)
```

```
/* Block signals (auto storage) */
typedef struct {
  real32_T Gain;                       /* '<Root>/Gain' */
} B_rtwdemo_basicsc_T;
```

The structure that contains block parameter data defines one field, `K1`, which represents the parameter object `K1`.

```
rtwdemodbtype(file,'/* Parameters (auto storage) */',...
    '/* Real-time Model Data Structure */',1,0)
```

```
/* Parameters (auto storage) */
struct P_rtwdemo_basicsc_T_ {
  int8_T K1;                           /* Variable: K1
                                        * Referenced by: '<Root>/Gain'
                                        */
};
```

Access Data Through Generated Interfaces

You can configure the generated code to contain extra code and files so that you can access model data through standardized interfaces. For example, use the C API to log signal data and tune parameters during execution.

Copy this custom source code into a file named `myHandCode.c` in your current folder.

```
#include "myHandHdr.h"

#define paramIdx 0 /* Index of the target parameter,
determined by inspecting the array of structures generated by the C API. */
#define sigIdx 0 /* Index of the target signal,
determined by inspecting the array of structures generated by the C API. */

void tuneFcn(rtwCAPI_ModelMappingInfo *mmi, time_T *tPtr)
{
    /* Take action with the parameter value only at
       the beginning of simulation and at the 5-second mark. */
    if (*tPtr == 0 || *tPtr == 5) {

        /* Local variables to store information extracted from
           the model mapping information (mmi). */
        void** dataAddrMap;
        const rtwCAPI_DataTypeMap *dataTypeMap;
        const rtwCAPI_ModelParameters *params;
        int_T addrIdx;
        uint16_T dTypeIdx;
        uint8_T slDataType;

        /* Use built-in C API macros to extract information. */
        dataAddrMap = rtwCAPI_GetDataAddressMap(mmi);
        dataTypeMap = rtwCAPI_GetDataTypeMap(mmi);
        params = rtwCAPI_GetModelParameters(mmi);
        addrIdx = rtwCAPI_GetModelParameterAddrIdx(params,paramIdx);
        dTypeIdx =  rtwCAPI_GetModelParameterDataTypeIdx(params,paramIdx);
        slDataType = rtwCAPI_GetDataTypeSLId(dataTypeMap, dTypeIdx);

        /* Handle data types 'double' and 'int8'. */
        switch (slDataType) {

            case SS_DOUBLE: {
                real_T* dataAddress;
```

```
                dataAddress = dataAddrMap[addrIdx];
                /* At the 5-second mark, increment the parameter value by 1. */
                if (*tPtr == 5) {
                    (*dataAddress)++;
                }
                printf("Parameter value is %f\n", *dataAddress);
                break;
            }

            case SS_INT8: {
                int8_T* dataAddress;
                dataAddress = dataAddrMap[addrIdx];
                if (*tPtr == 5) {
                    (*dataAddress)++;
                }
                printf("Parameter value is %i\n", *dataAddress);
                break;
            }
        }
    }
}

void logFcn(rtwCAPI_ModelMappingInfo *mmi, time_T *tPtr)
{
    /* Take action with the signal value only when
       the simulation time is an integer value. */
    if (*tPtr-(int_T)*tPtr == 0) {

        /* Local variables to store information extracted from
           the model mapping information (mmi). */
        void** dataAddrMap;
        const rtwCAPI_DataTypeMap *dataTypeMap;
        const rtwCAPI_Signals *sigs;
        int_T addrIdx;
        uint16_T dTypeIdx;
        uint8_T slDataType;

        /* Use built-in C API macros to extract information. */
        dataAddrMap = rtwCAPI_GetDataAddressMap(mmi);
        dataTypeMap = rtwCAPI_GetDataTypeMap(mmi);
        sigs = rtwCAPI_GetSignals(mmi);
        addrIdx = rtwCAPI_GetSignalAddrIdx(sigs,sigIdx);
        dTypeIdx =  rtwCAPI_GetSignalDataTypeIdx(sigs,sigIdx);
        slDataType = rtwCAPI_GetDataTypeSLId(dataTypeMap, dTypeIdx);
```

```
        /* Handle data types 'double' and 'single'. */
        switch (slDataType) {

            case SS_DOUBLE: {
                real_T* dataAddress;
                dataAddress = dataAddrMap[addrIdx];
                printf("Signal value is %f\n", *dataAddress);
                break;
            }

            case SS_SINGLE: {
                real32_T* dataAddress;
                dataAddress = dataAddrMap[addrIdx];
                printf("Signal value is %f\n", *dataAddress);
                break;
            }
        }
    }
}
```

Copy this custom header code into a file named `myHandHdr.h` in your current folder.

```
#include <stdio.h>
#include <string.h>
#include <math.h>
/* Include rtw_modelmap.h for definitions of C API macros. */
#include "rtw_modelmap.h"
#include "builtin_typeid_types.h"
#include "rtwtypes.h"
void tuneFcn(rtwCAPI_ModelMappingInfo *mmi, time_T *tPtr);
void logFcn(rtwCAPI_ModelMappingInfo *mmi, time_T *tPtr);
```

These files use the C API to access signal and parameter data in the code that you generate from the example model.

In the model, set **Configuration Parameters > Code Generation > Custom Code > Insert custom C code in generated > Header file** to `#include "myHandHdr.h"`. In the same pane in the Configuration Parameters dialog box, set **Additional Build Information > Source files** to `myHandCode.c`.

```
set_param('rtwdemo_basicsc','CustomHeaderCode','#include "myHandHdr.h"')
set_param('rtwdemo_basicsc','CustomSource','myHandCode.c')
```

Select **Configuration Parameters > All Parameters > MAT-file Logging**. The generated executable runs only until the simulation stop time (which you set in the model configuration parameters).

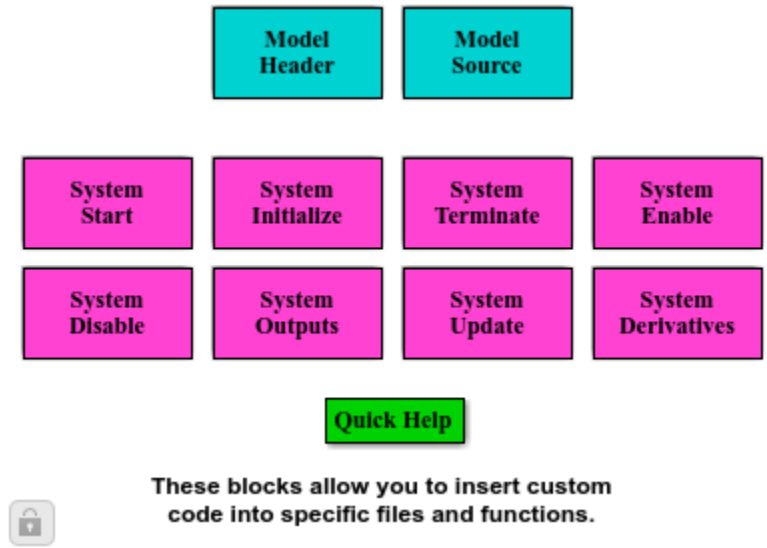
```
set_param('rtwdemo_basicsc','MatFileLogging','on')
```

Select all of the options under **Configuration Parameters > Code Generation > Interface > Generate C API for**.

```
set_param('rtwdemo_basicsc','RTWCAPIParams','on')
set_param('rtwdemo_basicsc','RTWCAPISignals','on')
set_param('rtwdemo_basicsc','RTWCAPIStates','on')
set_param('rtwdemo_basicsc','RTWCAPIRootIO','on')
```

Load the Custom Code block library.

```
custcode
```



Add a System Outputs block to the model.

```
add_block('custcode/System Outputs','rtwdemo_basicsc/System Outputs')
```

In the System Outputs block dialog box, set **System Outputs Function Execution Code** to this custom code:

```
{
rtwdemo_basicsc_U.input2++;
rtwCAPI_ModelMappingInfo *MMI = &(rtmGetDataMapInfo(rtwdemo_basicsc_M).mmi);
tuneFcn(MMI, rtmGetTPtr(rtwdemo_basicsc_M));
}
```

In the block dialog box, set **System Outputs Function Exit Code** to this custom code:

```
{
rtwCAPI_ModelMappingInfo *MMI = &(rtmGetDataMapInfo(rtwdemo_basicsc_M).mmi);
logFcn(MMI, rtmGetTPtr(rtwdemo_basicsc_M));
}
```

Alternatively, to configure the System Outputs block, at the command prompt, use these commands:

```
temp.TLCFile = 'custcode';
temp.Location = 'System Outputs Function';
temp.Middle = sprintf(['{\nrtwdemo_basicsc_U.input2++;'...
    '\nrtwCAPI_ModelMappingInfo *MMI = '...
    '&(rtmGetDataMapInfo(rtwdemo_basicsc_M).mmi);'...
    '\ntuneFcn(MMI, rtmGetTPtr(rtwdemo_basicsc_M));\n}']);
temp.Bottom = sprintf(['{\nrtwCAPI_ModelMappingInfo *MMI = '...
    '&(rtmGetDataMapInfo(rtwdemo_basicsc_M).mmi);'...
    '\nlogFcn(MMI, rtmGetTPtr(rtwdemo_basicsc_M));\n}']);
set_param('rtwdemo_basicsc/System Outputs','RTWdata',temp)
```

Generate code from the model.

```
rtwbuild('rtwdemo_basicsc')
```

```
### Starting build procedure for model: rtwdemo_basicsc
### Successful completion of build procedure for model: rtwdemo_basicsc
```

In the code generation report, view the interface file `rtwdemo_basicsc_capi.c`. This file initializes the arrays of structures that you can use to interact with data items through the C API. For example, in the array of structures `rtBlockSignals`, the first structure (index 0) describes the test-pointed output signal of the Gain block in the model.

```
file = fullfile('rtwdemo_basicsc_grt_rtw','rtwdemo_basicsc_capi.c');
rtwdemodbtype(file,'/* Block output signal information */',...
    '/* Individual block tuning',1,0)
```

```
/* Block output signal information */
static const rtwCAPI_Signals rtBlockSignals[] = {
  /* addrMapIndex, sysNum, blockPath,
   * signalName, portNumber, dataTypeIndex, dimIndex, fxpIndex, sTimeIndex
   */
  { 0, 0, TARGET_STRING("rtwdemo_basicsc/Gain"),
    TARGET_STRING(""), 0, 0, 0, 0, 0 },

  {
    0, 0, (NULL), (NULL), 0, 0, 0, 0, 0
  }
};
```

The fields of the structure, such as `addrMapIndex`, indicate indices into other arrays of structures, such as `rtDataAddrMap`, that describe the characteristics of the signal. These characteristics include the address of the signal data (a pointer to the data), the numeric data type, and the dimensions of the signal.

In the file `rtwdemo_basicsc.c`, view the code algorithm in the model `step` function. The algorithm first executes the custom code that you specified in the System Outputs block.

```
file = fullfile('rtwdemo_basicsc_grt_rtw','rtwdemo_basicsc.c');
rtwdemodbtype(file,'/* user code (Output function Body) */',...
    '/* Logic: ''<Root>/LogOp'' incorporates:',1,0)
```

```
  /* user code (Output function Body) */

  /* System '<Root>' */
  {
    rtwdemo_basicsc_U.input2++;
    rtwCAPI_ModelMappingInfo *MMI = &(rtmGetDataMapInfo(rtwdemo_basicsc_M).mmi);
    tuneFcn(MMI, rtmGetTPtr(rtwdemo_basicsc_M));
  }
```

This custom code first perturbs the input signal `input2` by incrementing the value of the signal each time the `step` function executes. The code then uses the built-in macro `rtmGetDataMapInfo` to extract model mapping information from the model data structure `rtwdemo_basicsc_M`. The pointer `MMI` points to the extracted mapping information, which allows the custom functions `tuneFcn` and `logFcn` to access the information contained in the arrays of structures that the C API file `rtwdemo_basicsc_capi.c` defines.

View the custom function `tuneFcn` in the file `myHandCode.c`. This function uses the C API (through the model mapping information `mmi`) and a pointer to the simulation time to print the value of the parameter `K1` at specific times during code execution. When the simulation time reaches 5 seconds, the function changes the parameter value in memory. By using a `switch case` block, the function can access the parameter data whether the data type is `int8` or `double`.

View the code algorithm in the model `step` function again. Near the end of the function, the algorithm executes the custom code that you specified in the System Outputs block. This code calls the custom function `logFcn`.

```
rtwdemodbtype(file,'/* user code (Output function Trailer) */',...
    '/* Matfile logging */',1,0)
```

```
  /* user code (Output function Trailer) */

  /* System '<Root>' */
  {
    rtwCAPI_ModelMappingInfo *MMI = &(rtmGetDataMapInfo(rtwdemo_basicsc_M).mmi);
    logFcn(MMI, rtmGetTPtr(rtwdemo_basicsc_M));
  }
```

View the custom function `logFcn` in the file `myHandCode.c`. The function uses the C API to print the value of the test-pointed signal. The function can access the signal data whether the data type is `single` or `double`.

At the command prompt, run the generated executable `rtwdemo_bascisc.exe`.

```
system('rtwdemo_basicsc')
```

The parameter and signal values appear in the Command Window output.

For more information about data interfaces, including the C API, see “Data Exchange Interfaces” (Simulink Coder).

See Also

Simulink.Parameter | `Simulink.Signal`

Related Examples

- “Default Data Structures in the Generated Code” on page 19-16

- "Configure Block Parameter Tunability for Rapid Prototyping" on page 19-56
- "Control Signals and States in Code by Applying Storage Classes" on page 19-123
- "Exchange Data Between Generated and External Code Using C API" (Simulink Coder)

Default Data Structures in the Generated Code

The generated code creates variables to represent model data such as signals, block parameters, and states. The code generation settings that you choose for a model determine the default scope of each datum. If the code generator applies a global scope to a datum, by default the datum appears as a field of a global data structure rather than a separate global variable. For example, the generated code creates default structures to contain block output signals, tunable parameters, and constant-valued nontunable parameters that the code generator cannot inline.

The table shows the most common global data structures in the generated code. The default name of each structure variable is *model_structname*. *model* is the name of the model. *structname* is the structure name in the table.

Global data structures generated for a standalone model

Structure Name	Data Represented in the Structure
`U`	Data from root Inport blocks
`Y`	Data from root Outport blocks
`B`	Block output signals
`ConstB`	Block outputs that have constant values
`P`	Block parameters
`DefaultP`	Default parameters in the system
`ConstP`	Constant parameters
`DW`	Discrete block states
`X`	Continuous block states
`XDot`	Derivatives of continuous states at each time step
`XDis`	Status of enabled subsystems
`ZCV`	Zero-crossing signals
`PrevZCX`	Previous zero-crossing signal states
`Obj`	Used by ERT C++ code generation to refer to referenced model objects

The table shows the most common global data structures generated for atomic subsystems and referenced models. The default name of each structure variable is *model_structname* for referenced models and *model_subsystem_structname* for subsystems.

Global data structures generated for subsystems and referenced models

Structure Name	Data Represented in the Structure
`B`	Block output signals
`ConstB`	Block outputs that have constant values
`P`	Block parameters
`DW`	Discrete block states
`MdlRefDW`	Discrete block states in referenced model
`X`	Continuous states in model reference
`XDis`	Status of enabled subsystems
`ZCV`	Zero-crossing signals
`RTM`	RT_Model structure

If you have an Embedded Coder license, you can control the names of these global structure variables. For more information, see "Global variables" (Simulink Coder) and "System-generated identifiers" (Simulink Coder).

You can exclude data from appearing in these structures by using:

- Storage classes. For example, you can use storage classes to represent signals, tunable parameters, and states as individual global variables. For more information, see "Control Signals and States in Code by Applying Storage Classes" (Simulink Coder) and "Block Parameter Representation in the Generated Code" (Simulink Coder).
- Configuration parameters, such as those on the **Optimization > Signals and Parameters** pane in the Configuration Parameters dialog box. You can adjust these configuration parameters to control the default representation of data. For more information, see "Optimization Pane: Signals and Parameters" (Simulink).

See Also

"Combine signal/state structures" (Simulink Coder)

Related Examples

- "Signal Representation in Generated Code" (Simulink Coder)
- "Block Parameter Representation in the Generated Code" (Simulink Coder)
- "Access Signal, State, and Parameter Data During Execution" on page 19-3

Use the Real-Time Model Data Structure

The code generator uses the real-time model (RT_MODEL) data structure. This structure is also referred to as the `rtModel` data structure. You can access `rtModel` data by using a set of macros analogous to the `ssSetxxx` and `ssGetxxx` macros that S-functions use to access `SimStruct` data, including noninlined S-functions compiled by the code generator.

You need to use the set of macros `rtmGetxxx` and `rtmSetxxx` to access the real-time model data structure. The `rtModel` is an optimized data structure that replaces `SimStruct` as the top level data structure for a model. The `rtmGetxxx` and `rtmSetxxx` macros are used in the generated code as well as from the `main.c` or `main.cpp` module. If you are customizing `main.c` or `main.cpp` (either a static file or a generated file), you need to use `rtmGetxxx` and `rtmSetxxx` instead of the `ssSetxxx` and `ssGetxxx` macros.

Usage of `rtmGetxxx` and `rtmSetxxx` macros is the same as for the `ssSetxxx` and `ssGetxxx` versions, except that you replace `SimStruct S` by real-time model data structure `rtM`. The following table lists `rtmGetxxx` and `rtmSetxxx` macros that are used in `grt_main.c` and `grt_main.cpp`.

Macros for Accessing the Real-Time Model Data Structure

rtm Macro Syntax	Description
`rtmGetdX(rtm)`	Get the derivatives of block continuous states
`rtmGetOffsetTimePtr(RT_MDL rtM)`	Return the pointer to vector that stores sample time offsets of the model associated with rtM
`rtmGetNumSampleTimes(RT_MDL rtM)`	Get the number of sample times that a block has
`rtmGetPerTaskSampleHitsPtr(RT_MDL)`	Return a pointer to `NumSampleTime` × `NumSampleTime` matrix
`rtmGetRTWExtModeInfo(RT_MDL rtM)`	Return an external mode information data structure of the model (used internally for external mode)
`rtmGetRTWLogInfo(RT_MDL)`	Return a data structure used by code generator logging (internal use only)
`rtmGetRTWRTModelMethodsInfo(RT_MDL)`	Return a data structure of real-time model methods information (internal use only)
`rtmGetRTWSolverInfo(RT_MDL)`	Return data structure containing solver information of the model (internal use only)

rtm Macro Syntax	Description
`rtmGetSampleHitPtr(RT_MDL)`	Return a pointer to Sample Hit flag vector
`rtmGetSampleTime(RT_MDL rtM, int TID)`	Get task sample time
`rtmGetSampleTimePtr(RT_MDL rtM)`	Get pointer to a task sample time
`rtmGetSampleTimeTaskIDPtr(RT_MDL rtM)`	Get pointer to a task ID
`rtmGetSimTimeStep(RT_MDL)`	Return simulation step type ID (`MINOR_TIME_STEP`, `MAJOR_TIME_STEP`)
`rtmGetStepSize(RT_MDL)`	Return the fundamental step size of the model
`rtmGetT(RT_MDL,t)`	Get the current simulation time
`rtmSetT(RT_MDL,t)`	Set the time of the next sample hit
`rtmGetTaskTime(RT_MDL,tid)`	Get the current time for the current task
`rtmGetTFinal(RT_MDL)`	Get the simulation stop time
`rtmSetTFinal(RT_MDL,finalT)`	Set the simulation stop time
`rtmGetTimingData(RT_MDL)`	Return a data structure used by timing engine of the model (internal use only)
`rtmGetTPtr(RT_MDL)`	Return a pointer to the current time
`rtmGetTStart(RT_MDL)`	Get the simulation start time
`rtmIsContinuousTask(rtm)`	Determine whether a task is continuous
`rtmIsMajorTimeStep(rtm)`	Determine whether the simulation is in a major step
`rtmIsSampleHit(RT_MDL,tid)`	Determine whether the sample time is hit
`rtmGetErrorStatus(rtm)`	Get the current error status
`rtmSetErrorStatus(rtm,val)`	Set the current error status
`rtmGetErrorStatusPointer(rtm)`	Return a pointer to the current error status
`rtmGetStopRequested(rtm)`	Return whether a stop is requested
`rtmGetBlockIO(rtm)`	Get the block I/O data structure
`rtmSetBlockIO(rtm,val)`	Set the block I/O data structure
`rtmGetContStates(rtm)`	Get the continuous states data structure
`rtmSetContStates(rtm,val)`	Set the continuous states data structure

rtm Macro Syntax	Description
`rtmGetDefaultParam(rtm)`	Get the default parameters data structure
`rtmSetDefaultParam(rtm,val)`	Set the default parameters data structure
`rtmGetPrevZCSigState(rtm)`	Get the previous zero-crossing signal state data structure
`rtmSetPrevZCSigState(rtm,val)`	Set the previous zero-crossing signal state data structure
`rtmGetRootDWork(rtm)`	Get the DWork data structure
`rtmSetRootDWork(rtm,val)`	Set the DWork data structure
`rtmGetU(rtm)`	Get the root inputs data structure (when root inputs are passed as part of the model data structure)
`rtmSetU(rtm,val)`	Set the root inputs data structure (when root inputs are passed as part of the model data structure)
`rtmGetY(rtm)`	Get the root outputs data structure (when root outputs are passed as part of the model data structure)
`rtmSetY(rtm,val)`	Set the root outputs data structure (when root outputs are passed as part of the model data structure)

For additional details on usage, see "SimStruct Macros and Functions Listed by Usage" (Simulink).

Related Examples

- "SimStruct Macros and Functions Listed by Usage" (Simulink)
- "Compare System Target File Support" (Simulink Coder)

Use Enumerated Data in Generated Code

In this section...

Enumerated Data Types

Enumerated data is data that is restricted to a finite set of values. An *enumerated data type* is a MATLAB class that defines a set of *enumerated values*. Each enumerated value consists of an *enumerated name* and an *underlying integer* which the software uses internally and in generated code. The following is a MATLAB class definition for an enumerated data type named `BasicColors`, which is used in the examples in this section.

```
classdef BasicColors < Simulink.IntEnumType
  enumeration
    Red(0)
    Yellow(1)
    Blue(2)
  end
end
```

For basic information about enumerated data types and their use in Simulink models, see “Use Enumerated Data in Simulink Models” (Simulink). For information about enumerated data types in Stateflow charts, see “Define Enumerated Data in a Chart” (Stateflow).

Specify Integer Data Type for Enumeration

When you specify a data type for your enumeration, you can:

- Control the size of enumerated data types in the generated code by specifying a superclass.

- Reduce RAM/ROM usage.
- Improve code portability.
- Improve integration with legacy code.

You can specify any of these integer data types:

- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `Simulink.IntEnumType`. Specify values in the range of the signed integer for your hardware platform.

Use a Class Definition in a MATLAB File

To specify an integer data type size, derive your enumeration class from the integer data type.

```
classdef Colors < int8
   enumeration
     Red(0)
     Green(1)
     Blue(2)
   end
end
```

The code generator generates this code:

```
typedef int8_T Colors;

#define Red      ((Colors)0)
#define Green ((Colors)1)
#define Blue     ((Colors)2)
```

Use the Function `Simulink.defineIntEnumType`

To specify an integer data type size, specify the name-value pair `StorageType` as the integer data type.

```
Simulink.defineIntEnumType('Colors',{'Red','Green','Blue'},...
[0;1;2],'StorageType','int8')
```

The code generator generates this code:

```
typedef int8_T Colors;

#define Red      ((Colors)0)
#define Green ((Colors)1)
#define Blue     ((Colors)2)
```

Customize Enumerated Data Type

When you generate code from a model that uses enumerated data, you can implement these static methods to customize the behavior of the type during simulation and in generated code:

- `getDefaultValue` — Specifies the default value of the enumerated data type.
- `getDescription` — Specifies a description of the enumerated data type.
- `getHeaderFile` — Specifies a header file where the type is defined for generated code.
- `getDataScope` — Specifies whether generated code exports or imports the enumerated data type definition to or from a separate header file.
- `addClassNameToEnumNames` — Specifies whether the class name becomes a prefix in generated code.

The first of these methods, `getDefaultValue`, is relevant to both simulation and code generation, and is described in "Specify a Default Enumerated Value" (Simulink). The other methods are relevant only to code generation. To customize the behavior of an enumerated type, include a version of the method in the `methods(Static)` section of the enumeration class definition. If you do not want to customize the type, omit the `methods(Static)` section. The table summarizes the methods and the data to supply for each one.

Static Method	Purpose	Default Value Without Implementing Method	Custom Return Value
`getDefaultValue`	Specifies the default enumeration member for the class.	First member specified in the enumeration definition	A character vector containing the name of an enumeration member in the class (see "Instantiate Enumerations" (Simulink)).

Static Method	Purpose	Default Value Without Implementing Method	Custom Return Value
`getDescription`	Specifies a description of the enumeration class.	`''`	A character vector containing the description of the type.
`getHeaderFile`	Specifies the name of a header file. The method `getDataScope` determines the significance of the file.	`''`	A character vector containing the name of the header file that defines the enumerated type. By default, the generated `#include` directive uses the preprocessor delimiter `"` instead of `<` and `>`. To generate the directive `#include <myTypes.h>`, specify the custom return value as `'<myTypes.h>'`.
`getDataScope`	Specifies whether generated code exports or imports the definition of the enumerated data type. Use the method `getHeaderFile` to specify the generated or included header file that defines the type.	`'Auto'`	One of: `'Auto'`, `'Exported'`, or `'Imported'`.
`addClassNameToEnumName`	Specifies whether to prefix the class name in generated code.	`false`	`true` or `false`.

Specify a Description

To specify a description for an enumerated data type, include this method in the `methods(Static)` section of the enumeration class:

```
function retVal = getDescription()
% GETDESCRIPTION  Optional description of the data type.
  retVal = 'description';
end
```

Substitute a MATLAB character vector for *`description`*. The generated code that defines the enumerated type includes the specified description.

Import Type Definition in Generated Code

To prevent generated code from defining an enumerated data type, which allows you to provide the definition in an external file, include these methods in the `methods(Static)` section of the enumeration class:

```
function retVal = getHeaderFile()
  % GETHEADERFILE Specifies the file that defines this type in generated code.
  % The method getDataScope determines the significance of the specified file.
  retVal = 'imported_enum_type.h';
end

function retVal = getDataScope()
  % GETDATASCOPE Specifies whether generated code imports or exports this type.
  % Return one of:
  % 'Auto':     define type in model_types.h, or import if header file specified
  % 'Exported': define type in a generated header file
  % 'Imported': import type definition from specified header file
  % If you do not define this method, DataScope is 'Auto' by default.
  retVal = 'Imported';
end
```

Instead of defining the type in *`model`*`_types.h`, which is the default behavior, generated code imports the definition from the specified header file using a `#include` statement like:

```
#include "imported_enum_type.h"
```

Generating code does not create the imported header file. You must provide the header file, using the file name specified by the method `getHeaderFile`, that defines the enumerated data type.

To create a Simulink enumeration that corresponds to your existing C-code enumeration, use the `Simulink.importExternalCTypes` function.

Export Type Definition in Generated Code

To generate a separate header file that defines an enumerated data type, include these methods in the `methods(Static)` section of the enumeration class:

```
function retVal = getDataScope()
  % GETDATASCOPE Specifies whether generated code imports or exports this type.
  % Return one of:
  % 'Auto':     define type in model_types.h, or import if header file specified
  % 'Exported': define type in a generated header file
  % 'Imported': import type definition from specified header file
  % If you do not define this method, DataScope is 'Auto' by default.
  retVal = 'Exported';
end

function retVal = getHeaderFile()
  % GETHEADERFILE Specifies the file that defines this type in generated code.
  % The method getDataScope determines the significance of the specified file.
  retVal = 'exported_enum_type.h';
end
```

Generated code exports the enumerated type definition to the generated header file `exported_enum_type.h`.

Add Prefixes To Class Names

By default, enumerated values in generated code have the same names that they have in the enumeration class definition. Alternatively, your code can prefix every enumerated value in an enumeration class with the name of the class. You can use this technique to prevent identifier conflicts or to improve the readability of the code. To specify class name prefixing, include this method in the `methods(Static)` section of an enumeration class:

```
function retVal = addClassNameToEnumNames()
  % ADDCLASSNAMETOENUMNAMES Specifies whether to add the class name
  % as a prefix to enumeration member names in generated code.
  % Return true or false.
  % If you do not define this method, no prefix is added.
  retVal = true;
end
```

Specify the return value as `true` to enable class name prefixing or as `false` to suppress prefixing. If you specify `true`, each enumerated value in the class appears in generated code as *`EnumTypeName_EnumName`*. For the example enumeration class `BasicColors` in "Enumerated Data Types" on page 19-22, the data type definition in generated code might look like this:

```
#ifndef _DEFINED_TYPEDEF_FOR_BasicColors_
#define _DEFINED_TYPEDEF_FOR_BasicColors_
```

```
typedef enum {
  BasicColors_Red = 0,            /* Default value */
  BasicColors_Yellow = 1,
  BasicColors_Blue = 2,
} BasicColors;

#endif
```

The enumeration class name `BasicColors` appears as a prefix for each of the enumerated names.

Control Enumerated Type Implementation in Generated Code

Suppose that you define an enumerated type `BasicColors`. You can specify that the generated code implement the type definition using:

- An `enum` block. The native integer type of your hardware is the underlying integer type for the enumeration members.
- A `typedef` statement and a series of `#define` macros. The `typedef` statement bases the enumerated type name on a specific integer data type, such as `int8`. The macros associate the enumeration members with the underlying integer values.

Implement Enumerated Type Using `enum` Block

To implement the type definition using an `enum` block:

- In Simulink, define the enumerated type using a `classdef` block in a script file. Derive the enumeration from the type `Simulink.IntEnumType`.
- Alternatively, use the function `Simulink.defineIntEnumType`. Do not specify the property `StorageType`.

When you generate code, the type definition appears in an `enum` block.

```
#ifndef _DEFINED_TYPEDEF_FOR_BasicColors_
#define _DEFINED_TYPEDEF_FOR_BasicColors_

typedef enum {
  Red = 0,            /* Default value */
  Yellow,
  Blue,
} BasicColors;
```

```
#endif
```

Implement Enumerated Type Using a Specific Integer Type

To implement the type definition using a `typedef` statement and `#define` macros:

- In Simulink, define the enumerated type using a `classdef` block in a script file. Derive the enumeration from a specific integer type such as `int8`.
- Alternatively, use the function `Simulink.defineIntEnumType`. Specify the property `StorageType` using a specific integer type such as `int8`.

When you generate code, the type definition appears as a `typedef` statement and a series of `#define` macros.

```
#ifndef _DEFINED_TYPEDEF_FOR_BasicColors_
#define _DEFINED_TYPEDEF_FOR_BasicColors_

typedef int8_T BasicColors;

#define Red ((BasicColors)0)            /* Default value */
#define Yellow ((BasicColors)1)
#define Blue ((BasicColors)2)

#endif
```

By default, the generated file *`model`*`_types.h` contains enumerated type definitions.

Type Casting for Enumerations

Safe Casting

A Simulink Data Type Conversion block accepts a signal of integer type. The block converts the input to one of the underlying values of an enumerated type.

If the input value does not match any of the underlying values of the enumerated type values, Simulink inserts a safe cast to replace the input value with the enumerated type default value.

Enable and Disable Safe Casting

You can enable or disable safe casting for enumerations during code generation for a Simulink Data Type Conversion block or a Stateflow block.

To control safe casting, enable or disable the **Saturate on integer overflow** block parameter. The parameter works as follows:

- **Enabled:** Simulink replaces a nonmatching input value with the default value of the enumerated values during simulation. The software generates a safe cast function during code generation.
- **Disabled:** For a nonmatching input value, Simulink generates an error during simulation. The software omits the safe cast function during code generation. In this case, the code is more efficient. However, the code may be more vulnerable to run-time errors.

Safe Cast Function in Generated Code

This example shows how the safe cast function `int32_T ET08_safe_cast_to_BasicColors` for the enumeration `BasicColors` appears in generated code when generated for 32-bit hardware.

```
static int32_T ET08_safe_cast_to_BasicColors(int32_T input)
{
 int32_T output;
 /* Initialize output value to default value for BasicColors (Red) */
 output = 0;
 if ((input >= 0) && (input <= 2)) {
 /* Set output value to input value if it is a member of BasicColors */
  output = input;
 }
 return output;
}
```

Through this function, the enumerated type's default value is used if the input value does not match one of underlying values of the enumerated type's values.

If the block's **Saturate on integer overflow** parameter is disabled, this function does not appear in generated code.

Enumerated Type Limitations

- Generated code does not support logging enumerated data.

See Also

`enumeration` | `Simulink.data.getEnumTypeInfo` | `Simulink.defineIntEnumType`

Related Examples

- "Use Enumerated Data in Simulink Models" (Simulink)
- "Simulink Enumerations" (Simulink)
- "Exchange Structured and Enumerated Data Between Generated and External Code" on page 21-28

Data Stores in Generated Code

In this section...

About Data Stores

A data store contains data that is accessible in a model hierarchy at or below the level in which the data store is defined. Data stores can allow subsystems and referenced models to share data without having to use I/O ports to pass the data from level to level. See "Data Stores with Data Store Memory Blocks" (Simulink) for information about data stores in Simulink. This section provides additional information about data store code generation.

Generate Code for Data Store Memory Blocks

To control the code generated for a Data Store Memory block, apply a storage class to the data store. You can associate a Data Store Memory block with a signal object that you store in a workspace or data dictionary, and control code generation for the block by applying the storage class to the object:

1. Instantiate the desired signal object.
2. Set the object's `CoderInfo.StorageClass` property to indicate the desired storage class.
3. Open the block dialog box for the Data Store Memory block that you want to associate with the signal object.
4. Enter the name of the signal object in the **Data store name** field.
5. Select **Data store name must resolve to Simulink signal object**.
6. *Do not* set the storage class field to a value other than `Auto` (the default).
7. Click **OK** or **Apply**.

Note When a Data Store Memory block is associated with a signal object, the mapping between the **Data store name** and the signal object name must be one-

to-one. If two or more identically named entities map to the same signal object, the name conflict is flagged as an error at code generation time. See "Resolve Conflicts in Configuration of Signal Objects" on page 19-131 for more information.

Storage Classes for Data Store Memory Blocks

You can control how Data Store Memory blocks in your model are stored and represented in the generated code by assigning storage classes and type qualifiers. You do this in almost exactly the same way you assign storage classes and type qualifiers for block states.

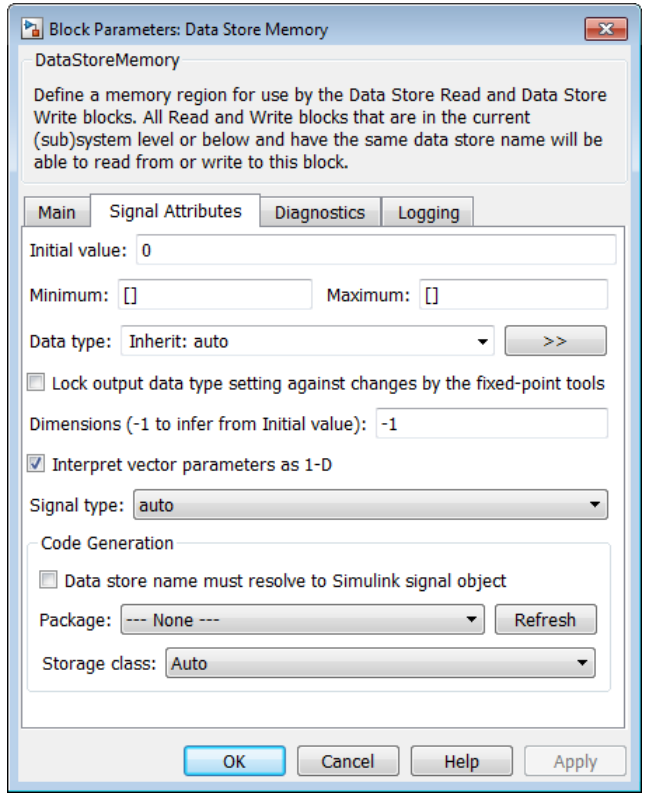
Data Store Memory blocks, like block states, have `Auto` storage class by default, and their memory is stored within the `DWork` vector. The symbolic name of the storage location is based on the data store name.

You can generate code from multiple Data Store Memory blocks that have the same data store name, subject to the following restriction: *at most one* of the identically named blocks can have a storage class other than `Auto`. An error is reported if this condition is not met.

For blocks with `Auto` storage class, the code generator produces a unique symbolic name for each block to avoid name clashes. For Data Store Memory blocks with storage classes other than `Auto`, the generated code uses the data store name as the symbol.

In the following model, a Data Store Write block writes to memory declared by the Data Store Memory block `myData`:

To control the storage declaration for a Data Store Memory block, use the **Code Generation > Signal object class** and **Code Generation > Storage class** drop-down lists of the Data Store Memory block dialog box. Set **Signal object class** to `Simulink.Signal` (the default), and choose a storage class from the **Storage class** drop-down list. The next figure shows the Data Store Memory block dialog box for the preceding model.



Data Store Memory blocks are nonvirtual because code is generated for their initialization in `.c` and `.cpp` files and their declarations in header files. The following table shows how the code generated for the Data Store Memory block in the preceding model differs for different settings of **Code Generation** > **Storage class**. The table gives the variable declarations and `MdlOutputs` code generated for the `myData` block.

Storage Class	Declaration	Code

`Auto` or `SimulinkGlobal`	In *`model`*`.h`
	
	`typedef struct`
	`D_Work_tag`
	`{`
	`  real_T myData;`
	`}`
	`D_Work;`
		`model``_DWork.myData =`
		`  rtb_SineWave;`

Storage Class	Declaration	Code

	In *model*.c or *model*.cpp
	`/* Block states (auto storage) */`
	`D_Work model_DWork;`	

`ExportedGlobal`	In *model*.c or *model*.cpp
	`/* Exported block states */`
	`real_T myData;`
	In *model*.h
	`extern real_T myData;`
		`myData = rtb_SineWave;`

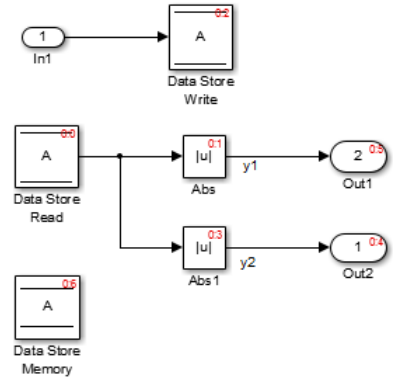
`ImportedExtern`	In *model*_private.h `extern real_T myData;`	`myData = rtb_SineWave;`
`ImportedExternPointer`	In *model*_private.h `extern real_T *myData;`	`(*myData) = rtb_SineWave;`

For information about applying storage classes, see "Control Signals and States in Code by Applying Storage Classes" (Simulink Coder).

Data Store Buffering in Generated Code

A Data Store Read block is a nonvirtual block that copies the value of the data store to its output buffer when it executes. Since the value is buffered, downstream blocks connected to the output of the data store read utilize the same value, even if a Data Store Write block updates the data store in between execution of two of the downstream blocks.

The next figure shows a model that uses blocks whose priorities have been modified to achieve a particular order of execution:




```
/* local block i/o variables */
real_T rtb_DataStoreRead;

/* DataStoreRead: '<Root>/Data Store Read' */
rtb_DataStoreRead = A;
```

Buffer the value of A

```
/* Abs: '<Root>/Abs1' incorporates:
 *  DataStoreRead: '<Root>/Data Store Read'
 */
y1 = fabs(A);
```

Use A (whose value equals the buffered value at this point

```
/* DataStoreWrite: '<Root>/Data Store Write' incorporates:
 *  Inport: '<Root>/In1'
 */
A = u1;
```

Update the value of A

```
/* Abs: '<Root>/Abs' */
y2 = fabs(rtb_DataStoreRead);
```

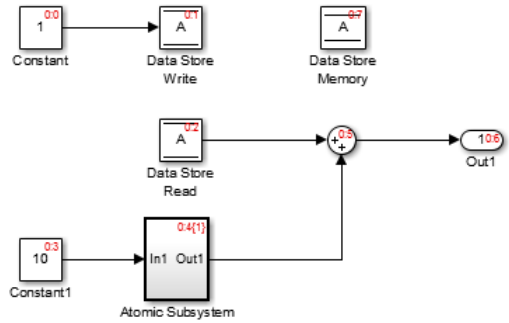
Consistently use the same buffered value as before the update to A

The following execution order applies:

1. The block Data Store Read buffers the current value of the data store A at its output.
2. The block Abs1 uses the buffered output of Data Store Read.
3. The block Data Store Write updates the data store.
4. The block Abs uses the buffered output of Data Store Read.

Because the output of Data Store Read is a buffer, both Abs and Abs1 use the same value: the value of the data store at the time that Data Store Read executes.

The next figure shows another example:




```
real_T rtb_DataStoreRead;

/* DataStoreWrite: '<Root>/Data Store Write' incorporates:
 *  Constant: '<Root>/Constant'
 */
A = DoBufferDSMRead2_P.Constant_Value;

/* DataStoreRead: '<Root>/Data Store Read' */
rtb_DataStoreRead = A;
```

Buffer the value of A

```
/* Outputs for atomic SubSystem: '<Root>/Atomic Subsystem' */
DoBufferDSMRead_AtomicSubsystem();
```

We don't do a global analysis to detect if this function writes to A

```
/* end of Outputs for SubSystem: '<Root>/Atomic Subsystem' */

/* Outport: '<Root>/Out1' incorporates:
 *  Sum: '<Root>/Sum'
 */
DoBufferDSMRead2_Y.Out1 = rtb_DataStoreRead + DoBufferDSMRead2_B.Abs;
```

Use the buffered value of A

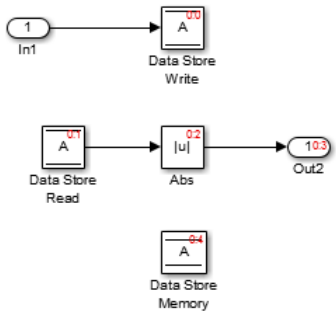
In this example, the following execution order applies:

1. The block Data Store Read buffers the current value of the data store A at its output.
2. Atomic Subsystem executes.
3. The Sum block adds the output of Atomic Subsystem to the output of Data Store Read.

Simulink assumes that Atomic Subsystem might update the data store, so Simulink buffers the data store. Atomic Subsystem executes after Data Store Read buffers its

output, and the buffer provides a way for the Sum block to use the value of the data store as it was when Data Store Read executed.

In some cases, the code generator determines that it can optimize away the output buffer for a Data Store Read block, and the generated code refers to the data store directly, rather than a buffered value of it. The next figure shows an example:




```
/* Model step function */
void DONTbufferDSMRead_step(void)
{
  /* DataStoreWrite: '<Root>/Data Store Write' incorporates:
   *  Inport: '<Root>/In1'
   */
  A = u1;

  /* Abs: '<Root>/Abs' incorporates:
   *  DataStoreRead: '<Root>/Data Store Read'
   */
  y2 = fabs(A);
}
```

In the generated code, the argument of the `fabs()` function is the data store `A` rather than a buffered value.

Related Examples

- "Control Signals and States in Code by Applying Storage Classes" (Simulink Coder)
- "Structures in Generated Code Using Data Stores" on page 19-39
- "When to Use a Data Store" (Simulink)
- "Generate Code That Dereferences Data from a Literal Memory Address" on page 23-83

Structures in Generated Code Using Data Stores

If you use more than one data store to provide global access to multiple signals in generated code, you can combine the signals into a single structure variable by using one data store. This combination of signal data can help you integrate the code generated from a model with other existing code that requires the data in a structure format.

This example shows how to store several model signals in a structure in generated code using a single data store. To store multiple signals in a data store, you configure the data store to accept a composite signal, such as a nonvirtual bus signal or an array of nonvirtual bus signals.

Explore Example Model

1. Open the example model ex_bus_struct_in_code.

   The model contains three subsystems that perform calculations on the inputs from the top level of the model. In each subsystem, a Data Store Memory block stores an intermediate calculated signal.

2. Generate code with the model. In the code generation report, view the file `ex_bus_struct_in_code.c`. The code defines a global variable for each data store.

   ```
   real_T BioBTURate;
   real_T CoalBTURate;
   real_T GasBTURate;
   ```

Suppose that you want to integrate code generated from the example model with other existing code. Suppose also that the existing code requires access to all of the data from the three data stores in a single structure variable. You can use a data store to assemble all of the target data in a structure in generated code.

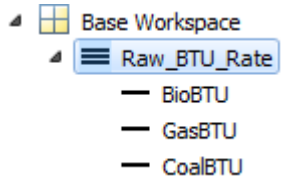
Configure Data Store

Configure a data store to contain multiple signals by creating a bus type to use as the data type of the data store. Define the bus type using the same hierarchy of elements as the structure that you want to appear in generated code.

1. Open the Bus Editor tool.

   ```
   buseditor
   ```

2 Define a new bus type `Raw_BTU_Rate` with one element for each of the three target signals. Name the elements `BioBTU`, `GasBTU`, and `CoalBTU`.



3 At the top level of the example model, add a Data Store Memory block.

4 In the block dialog box, set **Data store name** to `Raw_BTU_Data`. Click **Apply**.

5 On the **Signal Attributes** tab, set **Data type** to `Bus: Raw_BTU_Rate`.

6 Under **Code Generation**, set **Storage class** to `ExportedGlobal`.

Write to Data Store Elements

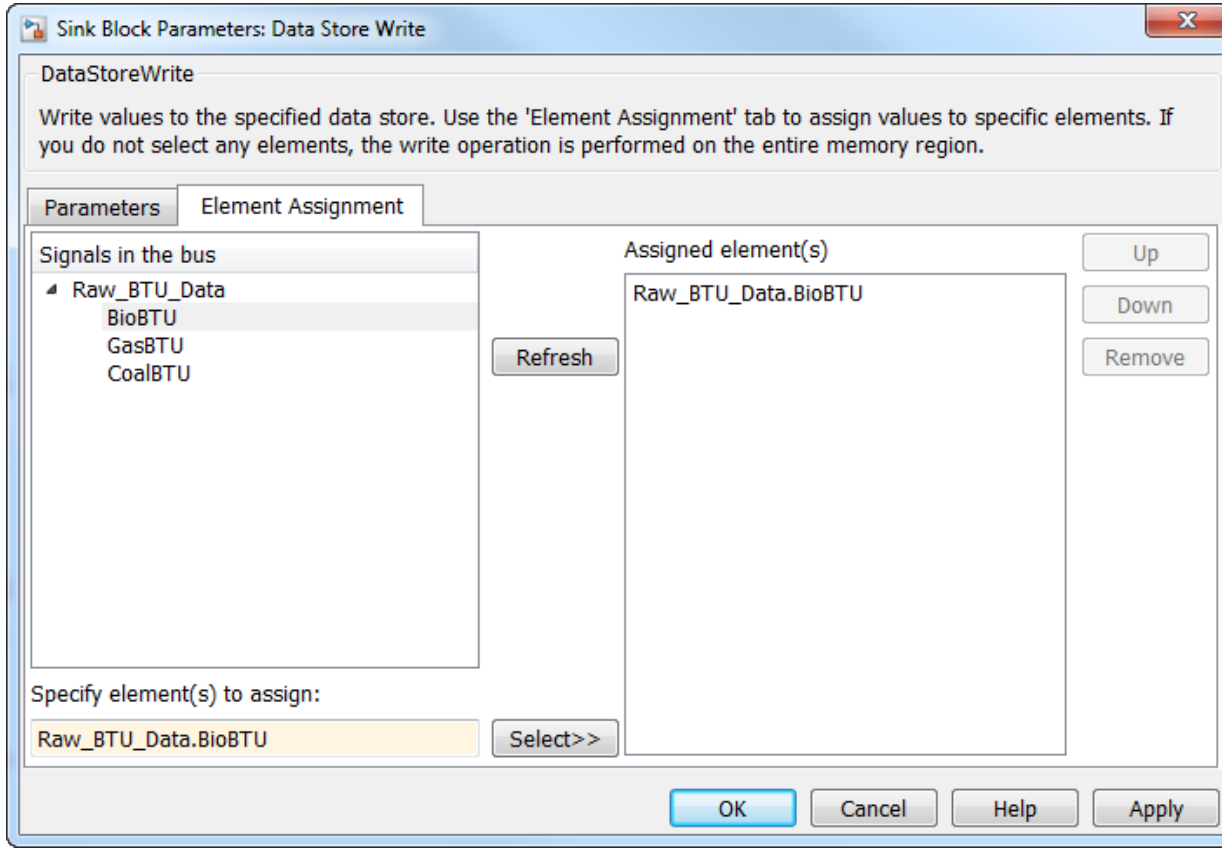
To write to a specific element of a data store, use a Data Store Write block. On the **Element Assignment** tab in the dialog box, you can specify to write to a single element, a collection of elements, or the entire contents of a data store.

1 Open the **Biomass Calc** subsystem.

2 Delete the Data Store Memory block `BioBTURate`.

3 In the block dialog box for the Data Store Write block, set **Data store name** to `Raw_BTU_Data`.

4 On the **Element Assignment** tab, under **Signals in the bus**, expand the contents of the data store `Raw_BTU_Data`. Click the element `BioBTU`, and then click **Select**. Click **OK**.



5 Modify the **Gas Calc** and **Coal Calc** subsystems similarly.

- Delete the Data Store Memory block in each subsystem.
- In each Data Store Write block dialog box, set **Data store name** to `Raw_BTU_Data`.
- In the **Gas Calc** subsystem, use the Data Store Write block to write to the data store element `GasBTU`. In the **Coal Calc** subsystem, write to the element `CoalBTU`.

Generate Code with Data Store Structure

1. Generate code for the example model.
2. In the code generation report, view the file `ex_bus_struct_in_code_types.h`. The code defines a structure that corresponds to the bus type `Raw_BTU_Rate`.

```
typedef struct {
  real_T BioBTU;
  real_T GasBTU;
  real_T CoalBTU;
} Raw_BTU_Rate;
```

3. View the file `ex_bus_struct_in_code.c`. The code represents the data store with a global variable `Raw_BTU_Data` of the structure type `Raw_BTU_Rate`. In the model step function, the code assigns the data from the calculated signals to the fields of the global variable `Raw_BTU_Data`.

See Also

Simulink.Bus | Data Store Write | Data Store Memory | Data Store Read

Related Examples

- "Data Stores with Buses and Arrays of Buses" (Simulink)
- "Access Data Stores with Simulink Blocks" (Simulink)
- "When to Use a Data Store" (Simulink)
- "About Data Stores" on page 19-32

Specify Single-Precision Data Type for Embedded Application

When you want code that uses only single precision, such as when you are targeting a single-precision processor, you can use model configuration parameters and block parameters to prevent the introduction of `double` in the model.

To design and validate a single-precision model, see "Validate a Floating-Point Embedded Model" (Simulink). If you have Fixed-Point Designer, you can use the Single Precision Converter app (see "Single-Precision Design for Simulink" (Fixed-Point Designer)).

Use `single` Data Type as Default for Underspecified Types

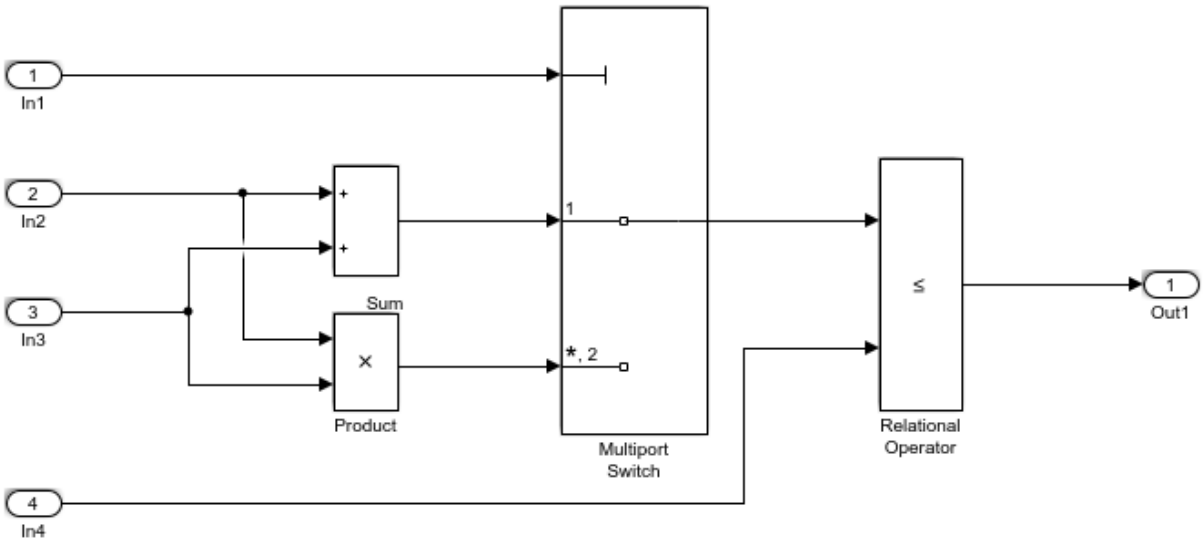
This example shows how to avoid introducing a double-precision data type in code generated for a single-precision hardware target.

If you specify an inherited data type for signals, but data type propagation rules cannot determine data types for the signals, the signal data types default to `double`. You can use a model configuration parameter to specify the default data type as `single`.

Explore Example Model

Open the example model rtwdemo_underspecified_datatype.

```
model = 'rtwdemo_underspecified_datatype';
open_system(model);
```



The root inports `In2`, `In3`, and `In4` specify `Inherit: Auto` for the **Data type** block parameter. The downstream blocks also use inherited data types.

Generate Code with `double` as Default Data Type

The model starts with the configuration parameter **System target file** set to `ert.tlc`, which requires Embedded Coder. Set **System target file** to `grt.tlc` instead.

```
set_param(model,'SystemTargetFile','grt.tlc')
```

Generate code from the model.

```
rtwbuild(model)
```

```
### Starting build procedure for model: rtwdemo_underspecified_datatype
### Successful completion of build procedure for model: rtwdemo_underspecified_datatype
```

In the code generation report, view the file `rtwdemo_underspecified_datatype.h`. The code uses the `double` data type to define the variables `In2`, `In3`, and `In4` because the Inport data types are underspecified in the model.

```
cfile = fullfile('rtwdemo_underspecified_datatype_grt_rtw',...
    'rtwdemo_underspecified_datatype.h');
rtwdemodbtype(cfile,...
    '/* External inputs (root inport signals with auto storage) */',...
    '/* External outputs (root outports fed by signals with auto storage) */', 1, 0);
```

```
/* External inputs (root inport signals with auto storage) */
typedef struct {
  int8_T In1;                          /* '<Root>/In1' */
  real_T In2;                          /* '<Root>/In2' */
  real_T In3;                          /* '<Root>/In3' */
  real_T In4;                          /* '<Root>/In4' */
} ExtU_rtwdemo_underspecified_d_T;
```

Generate Code with `single` as Default Data Type

Open the Configuration Parameters dialog box. On the **Optimization** pane, select `single` in the **Default for underspecified data type** drop-down list.

Alternatively, enable the optimization at the command prompt.

```
set_param(model, 'DefaultUnderspecifiedDataType', 'single');
```

Generate code from the model.

```
rtwbuild(model)
```

```
### Starting build procedure for model: rtwdemo_underspecified_datatype
### Successful completion of build procedure for model: rtwdemo_underspecified_datatype
```

In the code generation report, view the file `rtwdemo_underspecified_datatype.h`. The code uses the `single` data type to define the variables `In2`, `In3`, and `In4`.

```
rtwdemodbtype(cfile,...
    '/* External inputs (root inport signals with auto storage) */',...
    '/* External outputs (root outports fed by signals with auto storage) */', 1, 0);
```

```
/* External inputs (root inport signals with auto storage) */
typedef struct {
```

```
  int8_T In1;                          /* '<Root>/In1' */
  real32_T In2;                        /* '<Root>/In2' */
  real32_T In3;                        /* '<Root>/In3' */
  real32_T In4;                        /* '<Root>/In4' */
} ExtU_rtwdemo_underspecified_d_T;
```

Related Examples

- “Default for underspecified data type” (Simulink)
- “Subnormal Number Performance” on page 53-18
- “Standard math library” (Simulink Coder)
- “Data Type Replacement” on page 21-36
- “About Data Types in Simulink” (Simulink)

Block Parameter Representation in the Generated Code

Blocks have numeric parameters that determine how they calculate output values. For example, a Gain block has a **Gain** parameter. A Discrete Transfer Fcn has multiple parameters that represent the coefficients.

When you generate code from a model, block parameters can appear in the code as inlined numbers, formal parameters of functions, or global variables. Control parameter representation so that you can tune parameter values during algorithm execution, integrate the generated code with your handwritten code, and improve execution of the generated code.

For information about setting block parameter values in a model, see "Set Block Parameter Values" (Simulink).

Default Parameter Representation

You can control whether, by default, block parameters appear in the generated code as:

- Tunable fields of a global structure that contains parameter data. For example, if the value of the **Gain** parameter of a Gain block is `5.2`, the generated code algorithm can appear as `input = myModel_P.myGainBlock_Gain * output;` where the field `myModel_P.myGainBlock_Gain` is initialized to `5.2`.

  Use this configuration to enable you to change parameter data during execution for experimentation and rapid prototyping.
- Inlined numeric constants whose values you cannot change during execution. For example, the generated code algorithm can appear as `input = 5.2 * output;`.

  Use this configuration to optimize the generated code for production. See "Inline Numeric Values of Block Parameters" on page 53-43.

Configure Parameters as Tunable by Default

When you set **Configuration Parameters > Optimization > Signals and Parameters > Default parameter behavior** (see "Default parameter behavior" (Simulink)) to `Tunable` (the default when you use a GRT-based system target file such as `grt.tlc`), the generated code allocates memory to represent block parameters. Therefore, you can tune the block parameter values during code execution.

If you use numeric expressions, such as `3.57` or `5/2`, to specify block parameter values in the model, each block parameter appears in the generated code as a tunable field of the global parameters structure (for example, *`model`*`_P`). The code generator initializes each field by using the corresponding block parameter value from the model.

Workspace variables are variables that you use to specify block parameter values in a model. Workspace variables include numeric MATLAB variables and `Simulink.Parameter` objects that you store in a workspace, such as the base workspace, or in a data dictionary.

When you set **Default parameter behavior** to `Tunable`, workspace variables that use the storage class `Auto` appear in the generated code as tunable fields of the global parameters structure. If you use such a variable to specify multiple block parameter values, the variable appears as a single field of the global parameters structure. The code does not create multiple fields to represent the block parameters. Therefore, tuning the field value during code execution changes the mathematical behavior of the model in the same way as tuning the value of the MATLAB variable or parameter object during simulation.

For more information about the default global structures in the generated code that store signal, state, and parameter data, see "Default Data Structures in the Generated Code" on page 19-16.

To configure block parameter tunability for rapid prototyping, see "Configure Block Parameter Tunability for Rapid Prototyping" on page 19-56.

Configure Parameters as Inlined by Default

When you set **Default parameter behavior** to `Inlined` (the default when you use an ERT-based system target file such as `ert.tlc`), the generated code algorithm inlines the numeric values of block parameters. Therefore, you cannot tune the block parameters during code execution. Workspace variables that use the storage class `Auto`, such as numeric MATLAB variables, also appear in the generated code as inlined constants.

However, the code cannot represent some parameters, such as arrays, as inlined constants. These parameters appear as fields of a global structure that contains constant-valued, nontunable parameters. The structure uses the `const` type qualifier.

The code generator pools equivalent parameter values into a single field of the `const` structure instead of creating multiple fields. For example, if two Gain blocks use the nonscalar value `[2 5 7]` for the value of the **Gain** parameter, the code generator creates

a single structure field to store that value, and shares the field between the lines of code for each block. This pooling reduces the memory consumption of the structure and facilitates code reuse (see "Shared Constant Parameters for Code Reuse" on page 3-24).

If you have an Embedded Coder license, you can generate scalar inlined parameters as macros instead of literal numbers to improve code readability. See "Generate scalar inlined parameters as" (Simulink Coder).

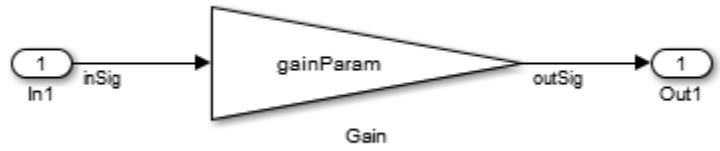
Override Default Parameter Behavior by Creating Global Variables in the Generated Code

You can control the code generated for a block parameter by applying a *storage class* or custom storage class to a `Simulink.Parameter` object. Use the parameter object to set the value of one or more block parameters in a model. By using this technique, you override the setting of **Default parameter behavior** for individual block parameters. For example, you can use this technique to import or export the corresponding generated variable to or from the generated code.

Choose from these built-in storage classes:

- `Auto`: The default storage class. Control the parameter tunability in the generated code by setting the model configuration parameter **Default parameter behavior**.
- `SimulinkGlobal`: Store the parameter object as a field of the generated global parameters structure, *`model`*`_P`, as if you set **Default parameter behavior** to `Tunable`. You can tune the parameter during code execution.
- `ExportedGlobal`: Export the declaration and definition of the parameter from the generated code as an individual global variable. You can tune the variable value during execution and use it in your custom code.
- `ImportedExtern`: Import the definition of the parameter from your custom code.
- `ImportedExternPointer`: Import a pointer to the parameter from your custom code.

For each of the storage classes, the table shows the declaration and the code generated for the workspace variable, `gainParam`, in the example model, `param_examp`. The numeric value of `gainParam` is `15.23`.



Storage Class	Declaration	Definition	Algorithmic Code
`Auto` (with **Default parameter behavior** set to `Inlined`)	None, because the parameter is inlined	None, because the parameter is inlined	`outSig = 15.23 * inSig;`

`SimulinkGl`	In *`model`*`.h`
	`struct P_param_examp_T_ {`
	`  real_T gainParam;`
	`};`
	`extern P_param_examp_T param_exa`
		In *`model`*`_data.c`
		`P_param_examp_T param_examp_P =`
		`  15.23`
		`};`
			`outSig =`
			` param_examp_P.gainParam *`
			` inSig;`

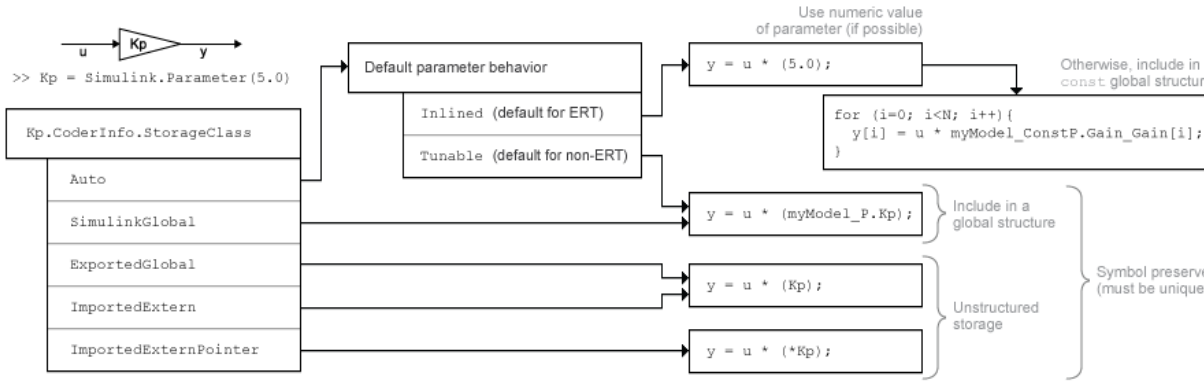
`ExportedGl`	In *`model`*`.h` `extern real_T gainParam;`	In *`model`*`.c` `real_T gainParam = 15.23;`	`outSig =` ` gainParam * inSig;`
`ImportedEx`	In *`model`*`_private.h` `extern real_T gainParam;`	None, because the parameter is imported	`outSig =` `  gainParam * inSig;`
`ImportedEx`	In *`model`*`_private.h` `extern real_T *gainParam;`	None, because the parameter is imported	`outSig =` `  *gainParam * inSig;`

For an example of how to use a storage class to control the code generated for a block parameter, see “Create Tunable Calibration Parameter in the Generated Code” on page 19-60.

If you have an Embedded Coder license, you can use and create custom storage classes for greater control over parameter representation. See “Introduction to Custom Storage Classes” on page 23-2.

Parameter Object Configuration Quick Reference Diagram

This diagram shows the code generation and storage class options that control the representation of parameter objects in the generated code.



Preservation of Expressions

You can specify block parameter values as expressions that use `Simulink.Parameter` objects or workspace variables. For example, you can use the expression `5 * gainParam`. See "Use MATLAB Functions and Custom Functions" (Simulink).

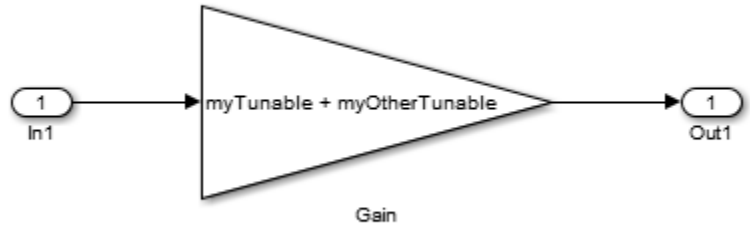
A *tunable workspace variable* is a `Simulink.Parameter` object or workspace variable that appears tunable in the generated code. For example, an object or variable is tunable if you apply a storage class other than `Auto` or if you set **Default parameter behavior** to `Tunable`.

An expression that contains one or more tunable workspace variables, model arguments, or tunable mask parameters is called a *tunable expression*. The expression is tunable because the code generator attempts to preserve the expression in the code. Because the code generator preserves the expression, you can change the values of the parameter data during code execution.

Expression with Tunable Parameter Objects

This example shows how the code generator preserves an expression that you use to set the value of a block parameter.

Consider the model `ex_tunable_expressions`.



The variables `myTunable` and `myOtherTunable` are `Simulink.Parameter` objects that use these property values.

Parameter Object Name	`StorageClass` Property	`Value` Property
`myTunable`	`ExportedGlobal`	`0.75`
`myOtherTunable`	`ExportedGlobal`	`2.0`

The generated code represents the tunable parameter objects `myTunable` and `myOtherTunable` as individual global variables. The algorithm preserves the expression `myTunable + myOtherTunable`.

```
ex_tunable_expressions_Y.Out1 = (myTunable + myOtherTunable) *
ex_tunable_expressions_U.In1;
```

Loss of Parameter Tunability

A block parameter, MATLAB variable, or `Simulink.Parameter` object is tunable if it appears in the generated code as data stored in memory, such as a global variable. For example, when you apply the storage class `ExportedGlobal` to a parameter object, the parameter object appears tunable in the generated code. When you set **Default parameter behavior** to `Tunable`, MATLAB variables and parameter objects appear tunable in the generated code. By definition, model arguments also appear tunable.

Under certain conditions, the code generator cannot maintain tunability of a parameter, variable, object, or expression. In this case, the code generator inlines the numeric value, preventing you from changing the value during code execution.

To detect these conditions in your model, set the model configuration parameter **Detect loss of tunability** (see "Detect loss of tunability" (Simulink)) to `warning` or `error`.

Tunable Expression Limitations

The code generator reduces certain expressions to an inlined numeric value in the generated code. For example, in "Expression with Tunable Parameter Objects" on page 19-51, you used the expression `myTunable + myOtherTunable` to set the value of a block parameter. If you instead use the expression `myTunable ^ myOtherTunable`, the code generator:

1. Evaluates the numeric value of the expression by using the values of the parameter objects. In this case, the expression value is $\text{myTunable}^{\text{myOtherTunable}} = 0.75^{2.0} = 0.5625$.
2. Inlines the expression value in the generated code algorithm. In this case, the calculation appears in the code as `ex_tunable_expressions_Y.Out1 = 0.5625 * ex_tunable_expressions_U.In1;`.

The code generator removes the tunability of `myTunable` and `myOtherTunable`.

To avoid loss of tunability due to unsupported expressions, observe these guidelines:

- Expressions involving complex (`i`) workspace variables or parameter objects are not supported.
- Certain operators and functions cause the code generator to reduce expressions and remove tunability. To determine whether an operator or function causes loss of tunability, use the information in this table.

Category	Operators or Functions
1	`+ - .* ./ < > <= >= == ~= & |`
2	`* /`
3	`abs`, `acos`, `asin`, `atan`, `atan2`, `boolean`, `ceil`, `cos`, `cosh`, `exp`, `floor`, `log`, `log10`, `sign`, `sin`, `sinh`, `sqrt`, `tan`, `tanh`, `single`, `int8`, `int16`, `int32`, `uint8`, `uint16`, `uint32`
4	`: .^ ^ [] {} . \ .\ ' .' , ;`

  - Use operators from category 1 without loss of tunability.
  - Use operators from category 2 in expressions as long as at least one operand is a scalar. For example, scalar/scalar and scalar/matrix operand combinations are supported, but matrix/matrix combinations are not supported.

    - You can use tunable workspace variables as arguments for the functions in category 3. If you use other functions, the code generator removes the tunability of the arguments.
    - The operators in category 4 are not supported.
- The Fcn and If blocks do not support tunable expressions for code generation or in referenced models.
- You can specify any data type for the `Simulink.Parameter` objects or workspace variables that makeup expressions. As long as the data type of these variables and objects and the data type of the corresponding block parameters are the same or `double`, the code generator can preserve tunability.

Linear Block Parameter Tunability

These blocks have a `Realization` parameter that affects the tunability of their numeric parameters:

- Transfer Fcn
- State-Space
- Discrete State-Space

To set the `Realization` parameter, you must use the command prompt:

```
set_param(gcb,'Realization','auto')
```

For the `Realization` parameter, you can choose these options:

- `general`: The block's numeric parameters appear tunable in the generated code.
- `sparse`: The generated code represents the block's parameters as transformed values that increase efficiency. The parameters are not tunable.
- `auto`: The default. If one or more of the block's parameters are tunable (for example, because you use a tunable parameter object to set a parameter value), then the block uses the `general` realization. Otherwise, the block uses the `sparse` realization.

  To tune the parameter values of one of these blocks during an external mode simulation, the block must use the `general` realization.

See Also

Simulink.Parameter

Related Examples

- "Configure Block Parameter Tunability for Rapid Prototyping" on page 19-56
- "Exchange and Reuse Parameter Data Between Generated Code and Existing Code" on page 23-11
- "Switch Between Sets of Parameter Values During Simulation and Code Execution" on page 19-103
- "Set Block Parameter Values" (Simulink)
- "Inline Numeric Values of Block Parameters" on page 53-43
- "Default Data Structures in the Generated Code" (Simulink Coder)

Configure Block Parameter Tunability for Rapid Prototyping

As you iteratively develop a model, to experiment with block parameter values (for example, the **Gain** parameter of a Gain block), you can *tune* the values during simulation or execution of the generated code. You can then observe the effect on signal values, and base your design decisions on analysis of these outputs. To access parameter data, you can configure the generated code to store the data in addressable global memory.

Optimization settings can make the generated code more efficient by inlining the numeric values of block parameters. To generate code that instead allocates addressable memory for parameters, you can disable the optimizations for all block parameters. Regardless of the settings that you use for the optimizations, you can also specify code generation settings for individual block parameters.

Goal	Considerations and More Information
Configure parameters as tunable by default	To prevent block parameter inlining in the generated code, and instead store parameter values in global memory, set the model configuration parameter **Default parameter behavior** to `Tunable`. Each block parameter appears in the generated code as a field of a global structure. For more information about **Default parameter behavior**, see "Default parameter behavior" (Simulink). For an example, see "Access Signal, State, and Parameter Data During Execution" (Simulink Coder).
Create separate global variables	When you set **Default parameter behavior** to `Tunable`, block parameters appear in the generated code as fields of a structure whose name you cannot explicitly specify. To instead store parameter data in separate global variables whose names, file placement, and other characteristics you can control, apply storage classes to `Simulink.Parameter` objects. See "Create Tunable Calibration Parameter in the Generated Code" on page 19-60.
Tune parameters during external mode simulation	When you generate code and an external executable from a model, you can simulate the model in external mode to communicate with the running executable. You can tune parameters and monitor signals during the simulation. However, in this simulation mode, tunability limitations that apply to code generation also apply to the simulation. For information about the code generation limitations, see "Loss of Parameter Tunability" on page 19-52.

Goal	Considerations and More Information
	For information about external mode, see "Set Up and Use Host/ Target Communication Channel" (Simulink Coder).
Tune parameters with Simulink Real-Time	If you have Simulink Real-Time, you can tune parameters and monitor signals during execution of your real-time application. Make parameters observable by using storage classes and the model configuration parameter **Default parameter behavior**. See "Tunable Block Parameters and MATLAB Variables" (Simulink Real-Time).

Related Examples

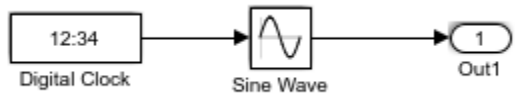
- "Deploy Algorithm Model for Real-Time Rapid Prototyping" on page 48-2
- "Access Signal, State, and Parameter Data During Execution" on page 19-3
- "Block Parameter Representation in the Generated Code" on page 19-47
- "Create Tunable Calibration Parameter in the Generated Code" on page 19-60
- "Switch Between Sets of Parameter Values During Simulation and Code Execution" on page 19-103

Tune Phase Parameter of Sine Wave Block During Code Execution

Under certain conditions, you cannot configure the **Phase** parameter of a Sine Wave block to appear in the generated code as a tunable global variable (for more information, see the block reference page). This example shows how to generate code so that you can tune the phase during execution.

Create the model `ex_phase_tunable`:

```
open_system('ex_phase_tunable')
```



Set **Default parameter behavior** to `Tunable` so that the parameters of the Sine Wave block appear in the generated code as tunable fields of the global parameter structure.

```
set_param('ex_phase_tunable','DefaultParameterBehavior','Tunable')
```

Generate code from the model.

```
rtwbuild('ex_phase_tunable')
```

```
### Starting build procedure for model: ex_phase_tunable
### Successful completion of code generation for model: ex_phase_tunable
```

In the code generation report, view the file `ex_phase_tunable.c`. The code algorithm in the model `step` function calculates the Sine Wave block output. The parameters of the block, including **Phase**, appear in the code as tunable structure fields.

```
file = fullfile('ex_phase_tunable_grt_rtw','ex_phase_tunable.c');
rtwdemodbtype(file,'/* Outport: ''<Root>/Out1'' incorporates:',...
    'ex_phase_tunable_P.SineWave_Bias;',1,1)
```

```
  /* Outport: '<Root>/Out1' incorporates:
   *  DigitalClock: '<Root>/Digital Clock'
   *  Sin: '<Root>/Sine Wave'
   */
  ex_phase_tunable_Y.Out1 = sin(ex_phase_tunable_P.SineWave_Freq *
    (((ex_phase_tunable_M->Timing.clockTick1+
```

```
   ex_phase_tunable_M->Timing.clockTickH1* 4294967296.0)) * 1.0) +
ex_phase_tunable_P.SineWave_Phase) * ex_phase_tunable_P.SineWave_Amp +
ex_phase_tunable_P.SineWave_Bias;
```

During code execution, you can assign new values to the structure field that corresponds to the **Phase** parameter.

Related Examples

- "Configure Block Parameter Tunability for Rapid Prototyping" on page 19-56

Create Tunable Calibration Parameter in the Generated Code

A *calibration parameter* is a value stored in global memory that an algorithm reads for use in calculations but does not write to. Calibration parameters are *tunable* because you can change the stored value during algorithm execution. You create calibration parameters so that you can:

- Determine an optimal parameter value by tuning the parameter and monitoring signal values during execution.
- Efficiently adapt an algorithm to different execution conditions by overwriting the parameter value stored in memory. For example, you can use the same control algorithm for multiple vehicles of different masses by storing different parameter values in each vehicle's engine control unit.

In Simulink, create a `Simulink.Parameter` object to represent a calibration parameter. You use the parameter object to set block parameter values, such as the **Gain** parameter of a Gain block. To control the representation of the parameter object in the generated code, you apply a storage class or custom storage class to the object.

To make block parameters accessible in the generated code by default, for example for rapid prototyping, set **Default parameter behavior** (see "Default parameter behavior" (Simulink)) to `Tunable`. See "Configure Block Parameter Tunability for Rapid Prototyping" on page 19-56.

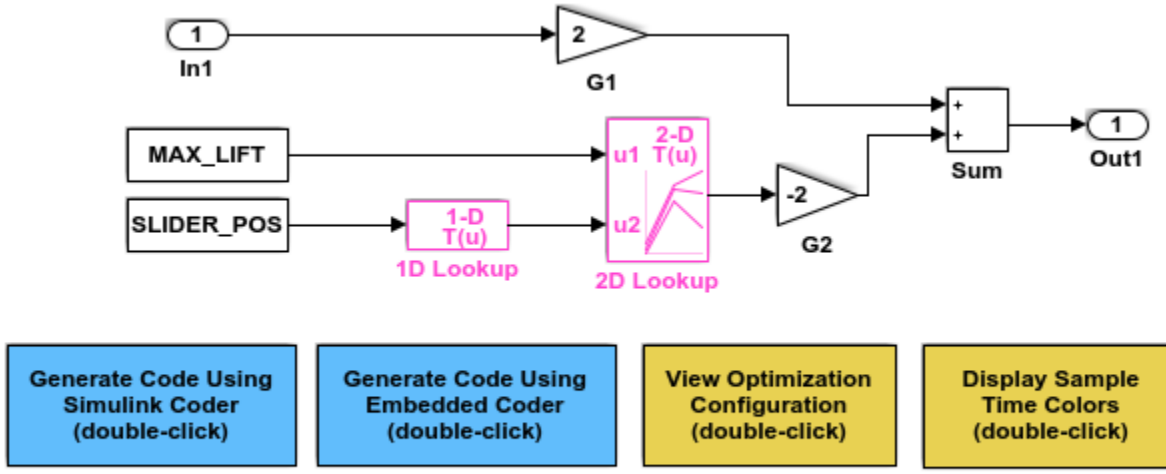
Represent Block Parameter as Tunable Global Variable

This example shows how to create tunable parameter data by representing a block parameter as a global variable in the generated code.

Configure Block Parameter by Using Parameter Object

Open the example model rtwdemo_paraminline.

```
rtwdemo_paraminline
```



In the G1 block dialog box, change **Gain** from `2` to `myGainParam`. Click **Apply**.

Click the action button next to the **Gain** parameter value. Select **Create Variable**.

In the Create New Data block dialog box, set **Value** to `Simulink.Parameter(2)`. Click **Create**. A `Simulink.Parameter` object `myGainParam` stores the parameter value, `2`, in the base workspace.

In the myGainParam dialog box, set **Storage class** to `ExportedGlobal` and click **OK**. This storage class causes the parameter object to appear in the generated code as a tunable global variable.

Alternatively, to create the parameter object and configure the model, use these commands at the command prompt:

```
set_param('rtwdemo_paraminline/G1','Gain','myGainParam')
myGainParam = Simulink.Parameter(2);
myGainParam.CoderInfo.StorageClass = 'ExportedGlobal';
```

Generate and Inspect Code

Generate code from the model.

```
rtwbuild('rtwdemo_paraminline')

### Starting build procedure for model: rtwdemo_paraminline
### Successful completion of build procedure for model: rtwdemo_paraminline
```

The generated file `rtwdemo_paraminline.h` contains an `extern` declaration of the global variable `myGainParam`. You can include (`#include`) this header file so that your code can read and write the value of the variable during execution.

```
file = fullfile('rtwdemo_paraminline_grt_rtw','rtwdemo_paraminline.h');
rtwdemodbtype(file,'extern real_T myGainParam;','extern real_T myGainParam;',1,1)


extern real_T myGainParam;             /* Variable: myGainParam
```

The file `rtwdemo_paraminline.c` allocates memory for and initializes `myGainParam`.

```
file = fullfile('rtwdemo_paraminline_grt_rtw','rtwdemo_paraminline.c');
rtwdemodbtype(file,'/* Exported block parameters */','real_T myGainParam = 2.0;',1,1)


/* Exported block parameters */
real_T myGainParam = 2.0;              /* Variable: myGainParam
```

The generated code algorithm in the model `step` function uses `myGainParam` for calculations.

```
rtwdemodbtype(file,'/* Model step function */','/* Model initialize function */',1,0)


/* Model step function */
void rtwdemo_paraminline_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Gain: '<Root>/G1'
   *  Inport: '<Root>/In1'
   *  Sum: '<Root>/Sum'
   */
  rtwdemo_paraminline_Y.Out1 = myGainParam * rtwdemo_paraminline_U.In1 + 150.0;
}
```

Configure Accessibility of Signal Data

When you tune the value of a parameter during algorithm execution, you monitor or capture output signal values to analyze the effect of the tuning. To represent signals in

the generated code as accessible data, you can use techniques such as test points and storage classes. See “Signal Representation in Generated Code” on page 19-112.

Programmatic Interfaces for Tuning Parameters

You can configure the generated code to include:

- A C application programming interface (API) for tuning parameters independent of external mode. The generated code includes extra code so that you can write your own code to access parameter values. See “Exchange Data Between Generated and External Code Using C API” (Simulink Coder).
- A Target Language Compiler API for tuning parameters independently of external mode. See “Parameter Functions” (Simulink Coder).

Set Tunable Parameter Minimum and Maximum Values

It is a best practice to specify minimum and maximum values for tunable parameters.

You can specify these minimum and maximum values:

- In the block dialog box that uses the parameter object. Use this technique to store the minimum and maximum information in the model.
- By using the properties of a Simulink.Parameter object that you use to set the parameter value. Use this technique to store the minimum and maximum information outside the model.

For more information, see “Specify Minimum and Maximum Values for Block Parameters” (Simulink).

Considerations for Other Modeling Goals

Goal	Considerations and More Information
Apply storage type qualifiers such as `const` and `volatile`	The storage type qualifier `const` can protect the integrity of parameter data and enable compilers to optimize machine code. If necessary, you can also use `volatile` to prevent compilers from eliminating storage for parameter data. If you have Embedded Coder, to generate storage type qualifiers, use custom storage classes. See “Type Qualifiers” on page 13-15.

Goal	Considerations and More Information
Generate ASAP2 description	You can generate an `a2l` file that uses the ASAP2 standard to describe your calibration parameters. For more information, see "Define ASAP2 Information for Parameters and Signals" on page 44-3.
Generate AUTOSAR (`arxml`) description	If you have an Embedded Coder license, you can generate an `arxml` file that describes calibration parameters used by models that you configure for the AUTOSAR standard. See "Model AUTOSAR Calibration Parameters and Lookup Tables".
Store lookup table data for calibration	To store lookup table data for calibration according to the ASAP2 or AUTOSAR standards (for example, STD_AXIS, COM_AXIS, or CURVE), you can use `Simulink.LookupTable` and `Simulink.Breakpoint` objects in lookup table blocks. However, some limitations apply. See Simulink.LookupTable. To work around the limitations of `Simulink.LookupTable` and `Simulink.Breakpoint` objects, use `Simulink.Parameter` and `AUTOSAR.Parameter` objects instead. See "Define ASAP2 Information for Parameters and Signals" on page 44-3 and "Configure STD_AXIS and COM_AXIS Lookup Tables for AUTOSAR Measurement and Calibration".
Use pragmas to store parameter data in specific memory locations	If you have an Embedded Coder license, to generate code that includes custom pragmas, use custom storage classes and memory sections. See "Control Data and Function Placement in Memory by Inserting Pragmas" on page 27-2.

See Also

Simulink.LookupTable | Simulink.Breakpoint | Simulink.Parameter

Related Examples

- "Exchange and Reuse Parameter Data Between Generated Code and Existing Code" on page 23-11
- "Reuse Parameter Data in Different Data Type Contexts" on page 19-93
- "Block Parameter Representation in the Generated Code" on page 19-47
- "Access Structured Data Through a Pointer That External Code Defines" on page 23-27

Specify Instance-Specific Parameter Values for Reusable Referenced Model

When you use model referencing to break a large system into components, each component is a separate model. You can reuse a component by referring to it with multiple Model blocks. Each Model block is an instance of the component. You can then configure a block parameter (such as the **Gain** parameter of a Gain block) to use either the same value or a different value for each instance of the component. To use different values, create and use a model argument to set the value of the block parameter.

When you generate code from a model hierarchy that uses model arguments, the arguments appear in the code as formal parameters of the referenced model entry-point functions, such as the output (`step`) function. The generated code then passes the instance-specific parameter values, which you specify in each Model block, to the corresponding function calls.

Whether you use or don't use model arguments, you can use storage classes to configure block parameters to appear in the generated code as tunable global variables. You can also use storage classes to generate tunable model argument values, which the generated code stores in memory and passes to the function calls. You can then change the values during execution.

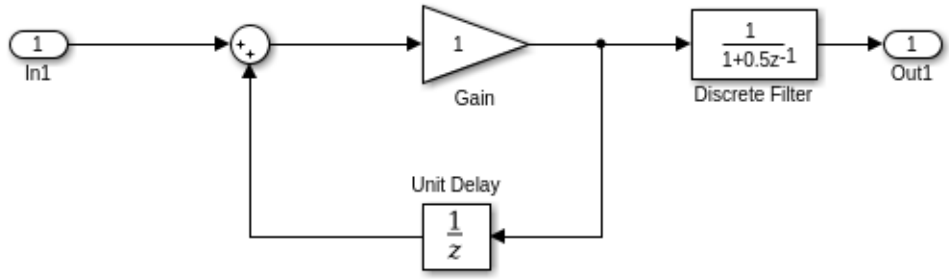
Pass Parameter Data to Referenced Model Entry-Point Functions as Arguments

Configure a referenced model to accept parameter data through formal parameters of the generated model entry-point function. This technique enables you to specify a different parameter value for each instance (Model block) of the referenced model.

Configure Referenced Model to Use Model Arguments

Create the model `ex_arg_code_ref`. This model represents a reusable algorithm.

```
open_system('ex_arg_code_ref')
```



In the Inport block dialog box, on the **Signal Attributes** tab, set **Data type** to `single`. Due to data type inheritance, the other signals in the model use the same data type.

```
set_param('ex_arg_code_ref/In1','OutDataTypeStr','single')
```

In the Model Explorer **Model Hierarchy** pane, expand the node **ex_arg_code_ref** and select **Model Workspace**.

Select **Add > Simulink Parameter** twice. Two `Simulink.Parameter` objects appear in the **Contents** pane.

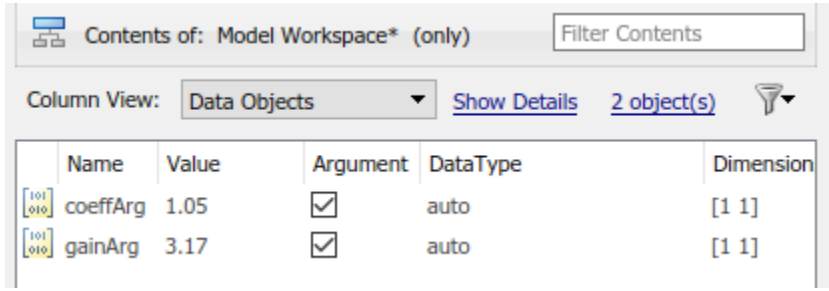
Rename the objects as `gainArg` and `coeffArg`.

Set the `Value` property of the objects. For example, set them to `3.17` and `1.05`, respectively.

```
modelWorkspace = get_param('ex_arg_code_ref','ModelWorkspace');
assignin(modelWorkspace,'gainArg',Simulink.Parameter(3.17));
assignin(modelWorkspace,'coeffArg',Simulink.Parameter(1.05));
```

In the **Contents** pane, for `coeffArg` and `gainArg`, select the check box in the **Argument** column.

```
set_param('ex_arg_code_ref','ParameterArgumentNames','coeffArg,gainArg')
```



In the `ex_arg_code_ref` model, in the Gain block dialog box, set **Gain** to `gainArg`.

```
set_param('ex_arg_code_ref/Gain','Gain','gainArg')
```

In the Discrete Filter block dialog box, set **Numerator** to `coeffArg`.

```
set_param('ex_arg_code_ref/Discrete Filter','Numerator','coeffArg')
```

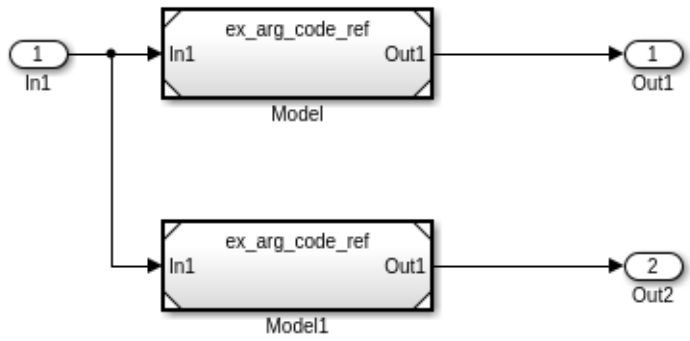
Save the `ex_arg_code_ref` model.

```
save_system('ex_arg_code_ref')
```

Specify Instance-Specific Parameter Values in Model Blocks

Create the model `ex_arg_code_ref`. This model uses multiple instances (Model blocks) of the reusable algorithm.

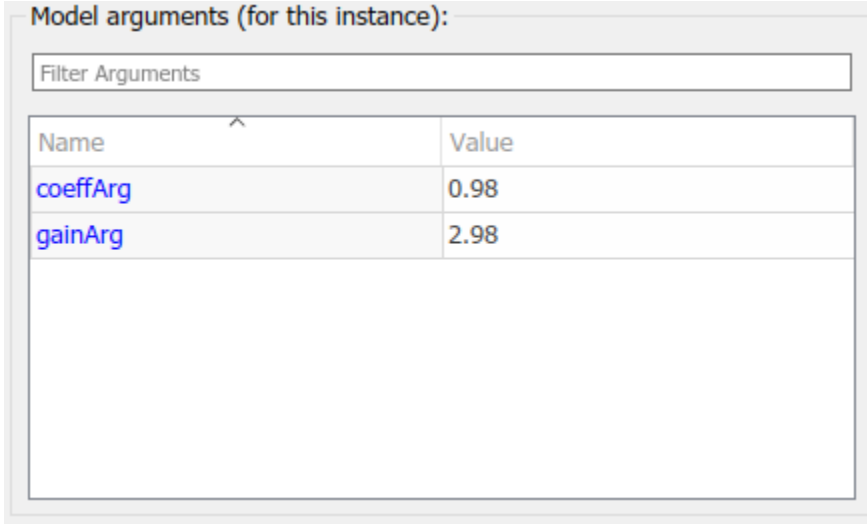
```
open_system('ex_arg_code')
```



Open the upper Model block dialog box.

Under **Model arguments**, set **coeffArg** and **gainArg** to `0.98` and `2.98`.

```
set_param('ex_arg_code/Model','ParameterArgumentValues',...
    struct('coeffArg','0.98','gainArg','2.98'))
```



In the other Model block dialog box, set **coeffArg** and **gainArg** to `1.11` and `3.34`.

```
set_param('ex_arg_code/Model1','ParameterArgumentValues',...
    struct('coeffArg','1.11','gainArg','3.34'))
```

Generate code from the top model.

```
rtwbuild('ex_arg_code')
```

The file `ex_arg_code_ref.c` defines the referenced model entry-point function, `ex_arg_code_ref`. The function has two formal parameters, `rtp_coeffArg` and `rtp_gainArg`, that correspond to the model arguments, `coeffArg` and `gainArg`. The formal parameters use the data type `real32_T`, which corresponds to the data type `single` in Simulink.

```
/* Output and update for referenced model: 'ex_arg_code_ref' */
void ex_arg_code_ref(const real32_T *rtu_In1, real32_T *rty_Out1,
```

```
                    DW_ex_arg_code_ref_f_T *localDW, real32_T rtp_coeffArg,
                    real32_T rtp_gainArg)
```

The file `ex_arg_code.c` contains the definition of the top model entry-point function, `ex_arg_code`. This function calls the referenced model entry-point function, `ex_arg_code_ref`, and uses the model argument values that you specified (such as `1.11` and `3.34`) as the values of `rtp_coeffArg` and `rtp_gainArg`.

```
/* ModelReference: '<Root>/Model' incorporates:
 *  Inport: '<Root>/In1'
 *  Outport: '<Root>/Out1'
 */
ex_arg_code_ref(&ex_arg_code_U.In1, &ex_arg_code_Y.Out1,
                &(ex_arg_code_DW.Model_DWORK1.rtdw), 0.98F, 2.98F);

/* ModelReference: '<Root>/Model1' incorporates:
 *  Inport: '<Root>/In1'
 *  Outport: '<Root>/Out2'
 */
ex_arg_code_ref(&ex_arg_code_U.In1, &ex_arg_code_Y.Out2,
                &(ex_arg_code_DW.Model1_DWORK1.rtdw), 1.11F, 3.34F);
```

The formal parameters use the data type `real32_T` (`single`) because:

1. The block parameters in `ex_arg_code_ref` determine their data types through internal rules. For example, in the Gain block dialog box, on the **Parameter Attributes** tab, **Parameter data type** is set to `Inherit: Inherit via internal rule` (the default). In this case, the internal rule chooses the same data type as the input and output signals, `single`.
2. The model arguments in the model workspace use context-sensitive data typing because the value of the `DataType` property is set to `auto` (the default). With this setting, the model arguments use the same data type as the block parameters, `single`.
3. The formal parameters in the generated code use the same data type as the model arguments, `single`.

Generate Tunable Argument Values

You can configure the instance-specific values in the Model blocks to appear in the generated code as tunable global variables. This technique enables you to store the parameter values for each instance in memory and to tune the values during code execution.

View the contents of the `ex_arg_code_ref` model workspace in Model Explorer.

Copy `gainArg` and `coeffArg` from the `ex_arg_code_ref` model workspace to the base workspace.

Rename `gainArg` as `gainForInst1`. Rename `coeffArg` as `coeffForInst1`.

```
gainForInst1 = getVariable(modelWorkspace,'gainArg');
gainForInst1 = copy(gainForInst1);
coeffForInst1 = getVariable(modelWorkspace,'coeffArg');
coeffForInst1 = copy(coeffForInst1);
```

Copy `gainForInst1` and `coeffForInst1` as `gainForInst2` and `coeffForInst2`.

```
gainForInst2 = copy(gainForInst1);
coeffForInst2 = copy(coeffForInst1);
```

Set the instance-specific parameter values by using the `Value` property of the parameter objects in the base workspace.

```
gainForInst1.Value = 2.98;
coeffForInst1.Value = 0.98;

gainForInst2.Value = 3.34;
coeffForInst2.Value = 1.11;
```

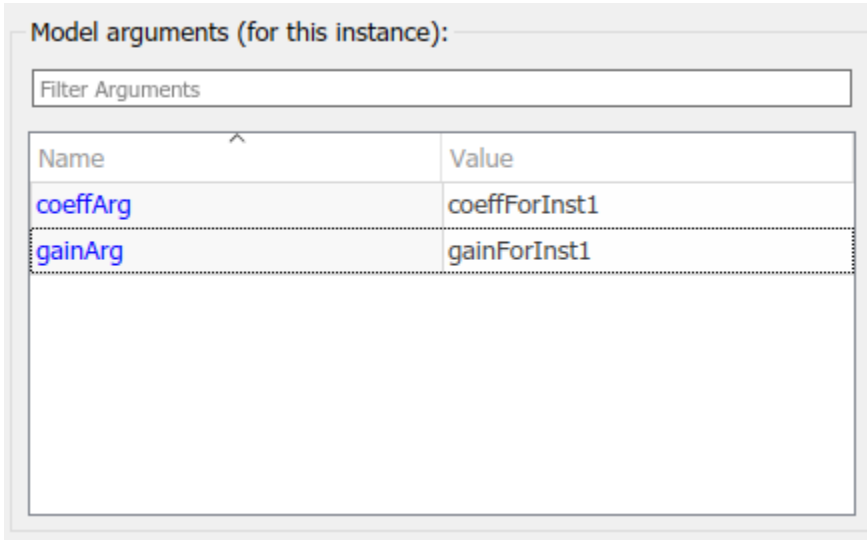
For the new parameter objects, set `StorageClass` to `ExportedGlobal`. This setting causes the parameter objects to appear in the generated code as tunable global variables.

```
gainForInst1.StorageClass = 'ExportedGlobal';
coeffForInst1.StorageClass = 'ExportedGlobal';
gainForInst2.StorageClass = 'ExportedGlobal';
coeffForInst2.StorageClass = 'ExportedGlobal';
```

In the top model, `ex_arg_code`, open the upper Model block dialog box.

Set **coeffArg** to `coeffForInst1` and **gainArg** to `gainForInst1`.

```
set_param('ex_arg_code/Model','ParameterArgumentValues',...
    struct('coeffArg','coeffForInst1','gainArg','gainForInst1'))
```



In the other Model block dialog box, set **coeffArg** to `coeffForInst2` and **gainArg** to `gainForInst2`.

```
set_param('ex_arg_code/Model1','ParameterArgumentValues',...
    struct('coeffArg','coeffForInst2','gainArg','gainForInst2'))
```

Generate code from the top model.

```
rtwbuild('ex_arg_code')
```

The file `ex_arg_code.c` defines the global variables that correspond to the parameter objects in the base workspace.

```
/* Exported block parameters */
real32_T coeffForInst1 = 0.98F;        /* Variable: coeffForInst1
                                        * Referenced by: '<Root>/Model'
                                        */
real32_T coeffForInst2 = 1.11F;        /* Variable: coeffForInst2
                                        * Referenced by: '<Root>/Model1'
                                        */
real32_T gainForInst1 = 2.98F;         /* Variable: gainForInst1
                                        * Referenced by: '<Root>/Model'
                                        */
real32_T gainForInst2 = 3.34F;         /* Variable: gainForInst2
                                        * Referenced by: '<Root>/Model1'
```

```
                                                                */
```

In each call to `ex_arg_code_ref`, the top model algorithm uses the global variables to set the values of the formal parameters.

```
/* ModelReference: '<Root>/Model' incorporates:
 *  Inport: '<Root>/In1'
 *  Outport: '<Root>/Out1'
 */
ex_arg_code_ref(&ex_arg_code_U.In1, &ex_arg_code_Y.Out1,
                &(ex_arg_code_DW.Model_DWORK1.rtdw), coeffForInst1,
                gainForInst1);

/* ModelReference: '<Root>/Model1' incorporates:
 *  Inport: '<Root>/In1'
 *  Outport: '<Root>/Out2'
 */
ex_arg_code_ref(&ex_arg_code_U.In1, &ex_arg_code_Y.Out2,
                &(ex_arg_code_DW.Model1_DWORK1.rtdw), coeffForInst2,
                gainForInst2);
```

The global variables in the generated code use the data type `real32_T` (`single`) because:

1. The parameter objects in the base workspace use context-sensitive data typing because the `DataType` property is set to `auto` (the default). With this setting, the parameter objects in the base workspace use the same data type as the model arguments, `single`.
2. The global variables in the generated code use the same data type as the parameter objects in the base workspace.

Group Multiple Model Arguments into Single Structure

Use the Model Explorer to copy `gainArg` and `coeffArg` from the `ex_arg_code_ref` model workspace into the base workspace.

```
temp = getVariable(modelWorkspace,'gainArg');
gainArg = copy(temp);
temp = getVariable(modelWorkspace,'coeffArg');
coeffArg = copy(temp);
```

At the command prompt, combine these two parameter objects into a structure, `structArg`.

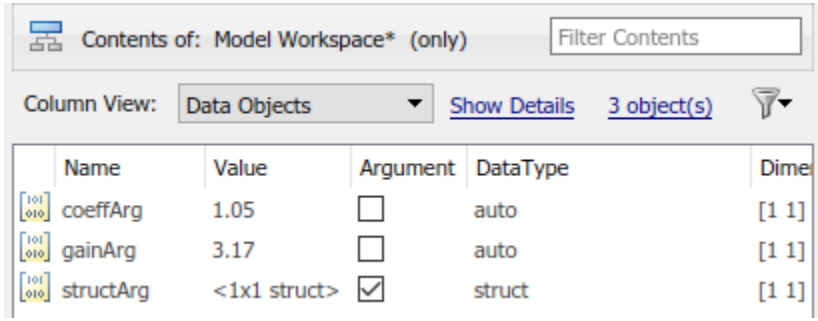
```
structArg = Simulink.Parameter(struct('gain',gainArg.Value,...
    'coeff',coeffArg.Value));
```

Use the Model Explorer to move `structArg` into the model workspace.

```
assignin(modelWorkspace,'structArg',copy(structArg));
clear structArg gainArg coeffArg
```

In the **Contents** pane, configure `structArg` as the only model argument.

```
set_param('ex_arg_code_ref','ParameterArgumentNames','structArg')
```



In the `ex_arg_code_ref` model, set the **Gain** parameter of the Gain block to `structArg.gain`.

```
set_param('ex_arg_code_ref/Gain','Gain','structArg.gain')
```

In the Discrete Filter block dialog box, set **Numerator** to `structArg.coeff`.

```
set_param('ex_arg_code_ref/Discrete Filter',...
    'Numerator','structArg.coeff')
```

At the command prompt, combine the four parameter objects in the base workspace into two structures. Each structure stores the parameter values for one instance of `ex_arg_code_ref`.

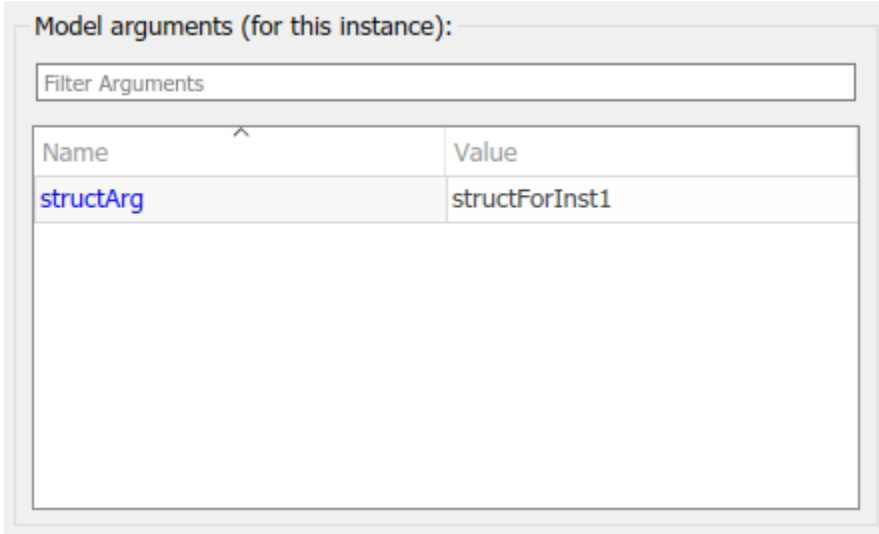
```
structForInst1 = Simulink.Parameter(struct('gain',gainForInst1.Value,...
    'coeff',coeffForInst1.Value));

structForInst2 = Simulink.Parameter(struct('gain',gainForInst2.Value,...
    'coeff',coeffForInst2.Value));
```

Apply the storage class `ExportedGlobal` to the parameter objects.

```
structForInst1.StorageClass = 'ExportedGlobal';
structForInst2.StorageClass = 'ExportedGlobal';
```

In the top model, in the upper Model block dialog box, set **structArg** to `structForInst1`.

```
set_param('ex_arg_code/Model','ParameterArgumentValues',...
    struct('structArg','structForInst1'))
```



In the other Model block dialog box, set **structArg** to `structForInst2`.

```
set_param('ex_arg_code/Model1','ParameterArgumentValues',...
    struct('structArg','structForInst2'))
```

Use the function `Simulink.Bus.createObject` to create a `Simulink.Bus` object. The hierarchy of elements in the object matches the hierarchy of the structure fields. The default name of the object is `slBus1`.

```
Simulink.Bus.createObject(structForInst1.Value);
```

Rename the bus object as `myParamStructType` by copying it.

```
myParamStructType = copy(slBus1);
```

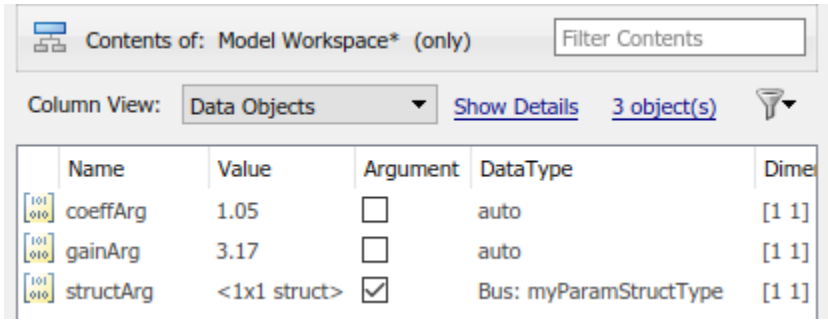
Set the data type of the parameter objects in the base workspace by using the bus object.

```
structForInst1.DataType = 'Bus: myParamStructType';
structForInst2.DataType = 'Bus: myParamStructType';
```

Use the Model Explorer to view the contents of the `ex_arg_code_ref` model workspace.

For the parameter object `structArg`, set `DataType` to `Bus: myParamStructType`.

```
temp = getVariable(modelWorkspace,'structArg');
temp = copy(temp);
temp.DataType = 'Bus: myParamStructType';
assignin(modelWorkspace,'structArg',copy(temp));
```



Save the `ex_arg_code_ref` model.

```
save_system('ex_arg_code_ref')
```

When you use structures to group parameter values, you cannot take advantage of context-sensitive data typing to control the data types of the fields of the structures (for example, the fields of `structForInst1`). However, you can use the properties of the bus object to control the field data types.

Set the data type of the elements in the bus object to `single`. The corresponding fields in the structures (such as `structForInst1` and `structArg`) use the same data type.

```
myParamStructType.Elements(1).DataType = 'single';
myParamStructType.Elements(2).DataType = 'single';
```

Generate code from the top model, `ex_arg_code`.

```
rtwbuild('ex_arg_code')
```

The file `ex_arg_code_types.h` defines the structure type `myParamStructType`, which corresponds to the `Simulink.Bus` object.

```
typedef struct {
  real32_T gain;
  real32_T coeff;
} myParamStructType;
```

In the file `ex_arg_code_ref.c`, the referenced model entry-point function has a formal parameter, `rtp_structArg`, that correspond to the model argument `structArg`.

```
/* Output and update for referenced model: 'ex_arg_code_ref' */
void ex_arg_code_ref(const real32_T *rtu_In1, real32_T *rty_Out1,
                     DW_ex_arg_code_ref_f_T *localDW, const myParamStructType
                     *rtp_structArg)
```

The file `ex_arg_code.c` defines the global structure variables that correspond to the parameter objects in the base workspace.

```
/* Exported block parameters */
myParamStructType structForInst1 = {
  2.98F,
  0.98F
} ;                                    /* Variable: structForInst1
                                        * Referenced by: '<Root>/Model'
                                        */

myParamStructType structForInst2 = {
  3.34F,
  1.11F
} ;                                    /* Variable: structForInst2
                                        * Referenced by: '<Root>/Model1'
                                        */
```

The top model algorithm in the file `ex_arg_code.c` passes the addresses of the structure variables to the referenced model entry-point function.

```
  /* ModelReference: '<Root>/Model' incorporates:
   *  Inport: '<Root>/In1'
   *  Outport: '<Root>/Out1'
   */
  ex_arg_code_ref(&ex_arg_code_U.In1, &ex_arg_code_Y.Out1,
                  &(ex_arg_code_DW.Model_DWORK1.rtdw), &structForInst1);

  /* ModelReference: '<Root>/Model1' incorporates:
   *  Inport: '<Root>/In1'
   *  Outport: '<Root>/Out2'
   */
```

```
ex_arg_code_ref(&ex_arg_code_U.In1, &ex_arg_code_Y.Out2,
                &(ex_arg_code_DW.Model1_DWORK1.rtdw), &structForInst2);
```

Control Data Types of Model Arguments and Argument Values

When you use model arguments, you can apply a data type to:

- The block parameters that use the arguments (for certain blocks, such as those in the Discrete library).
- The arguments in the referenced model workspace.
- The argument values that you specify in Model blocks.

To generate efficient code by eliminating unnecessary typecasts and C shifts, consider using inherited and context-sensitive data typing to match the data types.

- In the model workspace, use a MATLAB variable whose data type is `double` or a parameter object whose `DataType` property is set to `auto`. In this case, the variable or object uses the same data type as the block parameter.
- When you set the argument values in Model blocks, take advantage of context-sensitive data typing. To set an argument value, use an untyped value.

  - A literal number such as `15.23`. Do not use a typed expression such as `single(15.23)`.
  - A MATLAB variable whose data type is `double`.
  - A `Simulink.Parameter` object whose `DataType` property is set to `auto`.

  In these cases, the number, variable, or object uses the same data type as the model argument in the referenced model workspace. If you also configure the model argument to use context-sensitive data typing, you can control the data types of the block parameter, the argument, and the argument value by specifying the type only for the block parameter.

For basic information about controlling parameter data types, see “Parameter Data Types in the Generated Code” on page 19-79.

Use Model Argument in Different Data Type Contexts

If you use a model argument to set multiple block parameter values, and the data types of the block parameters differ, you cannot use context-sensitive data typing (`double` or `auto`) for the argument in the model workspace. You must explicitly specify a data type

for the argument. For example, if the argument in the model workspace is a parameter object (such as `Simulink.Parameter`), set the `DataType` property to a value other than `auto`. For more information about this situation, see “Reuse Parameter Data in Different Data Type Contexts” on page 19-93.

In this case, you can continue to take advantage of context-sensitive data typing to control the data type of the argument values that you specify in Model blocks. Each argument value uses the data type that you specify for the corresponding argument in the model workspace.

Related Examples

- “Code Generation of Referenced Models” on page 5-2
- “Block Parameter Representation in the Generated Code” on page 19-47
- “Parameterize Instances of a Reusable Referenced Model” (Simulink)
- “Organize Block Parameter Values into Structures in the Generated Code” on page 19-97
- “Parameter Data Types in the Generated Code” on page 19-79

Parameter Data Types in the Generated Code

The data type of a block parameter (such as the **Gain** parameter of a Gain block), numeric MATLAB variable, or `Simulink.Parameter` object determines the data type that the corresponding entity in the generated code uses (for example, a global variable or an argument of a function). To generate more efficient code, you can match parameter data types with signal data types or store parameters in smaller data types.

For basic information about setting block parameter data types in a model, see "Control Block Parameter Data Types" (Simulink).

Significance of Parameter Data Types

The data type that a block parameter, MATLAB variable, or parameter object uses determines the data type that the generated code uses to store the parameter value in memory. For example:

- If you set the model configuration parameter **Default parameter behavior** (see "Default parameter behavior" (Simulink)) to `Tunable`, the **Gain** parameter of a Gain block appears in the generated code as a field of a global structure that stores parameter data. If you apply the data type `single` to the block parameter in the model, the structure field uses the corresponding data type (`real32_T`).
- If you apply the storage class `ExportedGlobal` to a `Simulink.Parameter` object, the object appears in the generated code as a separate global variable. If you set the `DataType` property of the object to `int8`, the global variable in the code uses the corresponding data type (`int8_T`).
- If you configure a `Simulink.Parameter` object in a model workspace as a model argument, the object appears in the generated code as a formal parameter of a model entry-point function, such as the `step` function. The `DataType` property of the object determines the data type of the formal parameter.

Other than determining the data type that the generated code uses to store parameter values in memory, the data type of a parameter, variable, or object can also:

- Cause the block to cast the value of the parameter prior to code generation. The cast can result in overflow, underflow, or quantization.
- Cause the generated code to include extra code, for example saturation code.

Parameter Data Type Mismatch

When the data types of block parameters, workspace variables, and signals differ, blocks can use typecasts to reconcile the data type mismatches. These typecasts can cause the generated code algorithm, including the model `step` function, to include explicit casts to reconcile mismatches in storage data type and C bit shifts to reconcile mismatches in fixed-point scaling.

Parameter data type mismatches can occur when:

- The data type that you specify for a MATLAB variable or parameter object (`Simulink.Parameter`) differs from the data type of a block parameter. The block parameter typecasts the value of the variable or object.
- The data type that you specify for an initial value differs from the data type of the initialized signal or state.
- The data type that you specify for a block parameter differs from the data type of the signal or signals that the parameter operates on. Some blocks typecast the parameter to perform the operation. For example, the Gain block performs this typecast.

If you configure a variable or object to use bias or fractional fixed-point slope, the block parameter cannot perform the typecast. In this case, you must match the data type of the variable or parameter object with the data type of the block parameter. Use one of these techniques:

- Use context-sensitive data typing for the variable or parameter object. For a MATLAB variable, use a `double` number to set the value of the variable. For a parameter object, set the `DataType` property to `auto`.
- Use a `Simulink.AliasType` or `Simulink.NumericType` object to set the data type of the block parameter and the data type of a parameter object.

  Use this technique when you cannot rely on context-sensitive data typing, for example, when you use the field of a structure to set the value of the block parameter.
- Manually specify the same data type for the block parameter and the variable or parameter object.

  Use this technique to reduce the dependence of the model on inherited and context-sensitive data types and on external variables and objects.

For blocks that access parameter data through pointer or reference in the generated code, if you specify a different data type of the workspace variable and block parameter, the

generated code implicitly casts the data type of the variable to the data type of the block parameter. Note, that an implicit cast requires a data copy which could significantly increase RAM consumption and slow down code execution speed for large data sets. For example, Lookup Table blocks often access large vectors or matrices through pointer or reference in the generated code.

For information about matching parameter data types when you use model arguments, see “Control Data Types of Model Arguments and Argument Values” on page 19-77.

Detect Downcast and Loss of Precision due to Data Type Mismatches

You can configure diagnostic configuration parameters to detect unintentional data type mismatches that result in quantization and loss of parameter precision. See “Model Configuration Parameters: Data Validity Diagnostics” (Simulink).

Considerations for Other Modeling Patterns

When you use specific modeling patterns and constructs such as fixed-point data types, parameter structures, and lookup table objects, use different techniques to control parameter data types.

Tunable Parameters and Best-Precision Fixed-Point Scaling

To apply best-precision fixed-point scaling to a tunable block parameter or parameter object, you can use the Fixed-Point Tool to autoscale an entire system or use the Data Type Assistant to configure individual parameters or objects. See “Calculate Best-Precision Fixed-Point Scaling for Tunable Block Parameters” (Simulink).

If a tunable parameter uses best-precision fixed-point scaling, Simulink chooses a data type based on the minimum and maximum values that you specify for the parameter. You can specify these values in the block dialog box that uses the parameter or in the properties of a Simulink.Parameter object.

If you do not specify a minimum or maximum, Simulink chooses a data type based on the value of the parameter. The chosen scaling might restrict the range of possible tuning values. Therefore, it is a best practice to specify minimum and maximum values for each tunable parameter.

A tunable parameter can use best-precision scaling even if you do not specify it in the parameter data type. For example, the Gain block can choose a best-precision scaling if the **Parameter data type** in the block dialog box is set to `Inherit: Inherit via internal rule`. This setting is the default for the block.

Control Data Types of Structure Fields

When you use a structure as the value of a block parameter (for example to initialize a bus signal), or when you organize multiple block parameter values into a single structure, you can create a `Simulink.Bus` object to use as the data type of a `Simulink.Parameter` object. You can then control the data types of individual fields in the structure. See "Control Field Data Types and Characteristics by Creating Parameter Object" (Simulink) and "Control Data Types of Initial Condition Structure Fields" (Simulink).

Control Data Types of Lookup Table Objects

When you use Simulink.LookupTable and Simulink.Breakpoint objects to store table and breakpoint data for a lookup table block, to control the data types of the table and breakpoint data, use one of these techniques:

- Set the `Value` property of the embedded `Simulink.lookuptable.Table` and `Simulink.lookuptable.Breakpoint` objects by using untyped expressions such as `[1 2 3]`, which returns a `double` vector. To control the data type, set the `DataType` property to a value other than `auto`.

  Use this technique to separate the value of the table or breakpoint data from the data type, which can improve readability and understanding of your design. You can then use a `Simulink.NumericType` or `Simulink.AliasType` object to:

  - Customize the name of the data type in the generated code.
  - Match the data type of the table or breakpoint data with the data type of a signal in the model.
- Set the `Value` property of the embedded objects by using typed expressions such as `single([1 2 3])`. To use a fixed-point data type, set the `Value` property with an `fi` object.

  Set the `DataType` property of the embedded objects to the default value, `auto`. The table and breakpoint data then acquire the data type that you use to set the `Value` property.

Use this technique to store the data type information in the `Value` property, which can simplify the way you interact with the `Simulink.LookupTable` and `Simulink.Breakpoint` objects. You can leave the `DataType` property at the default value.

When you later change the breakpoint or table data in the `Value` property, preserve the data type information by using a typed expression. Alternatively, if you use a command at the command prompt or a script to change the data, to avoid using a typed expression, use subscripted assignment, `(:)`.

```
myLUTObject.Table.Value(:) = [4 5 6];
```

When you change the data stored in the `Value` property, if you do not use a typed expression or subscripted assignment, you lose the data type information.

When blocks in a subsystem use `Simulink.LookupTable` or `Simulink.Breakpoint` objects, you cannot set data type override (see "Control Fixed-Point Instrumentation and Data Type Override" (Simulink)) only on the subsystem. Instead, set data type override on the entire model.

Related Examples

- "Generate Efficient Code by Specifying Data Types for Block Parameters" on page 19-84
- "Reuse Parameter Data in Different Data Type Contexts" on page 19-93
- "Data Types Supported by Simulink" (Simulink)
- "Control Signal Data Types" (Simulink)
- "Block Parameter Representation in the Generated Code" on page 19-47

Generate Efficient Code by Specifying Data Types for Block Parameters

To generate more efficient code, match the data types of block parameters (such as the **Gain** parameter of a Gain block) with signal data types. Alternatively, you can store parameters in smaller data types.

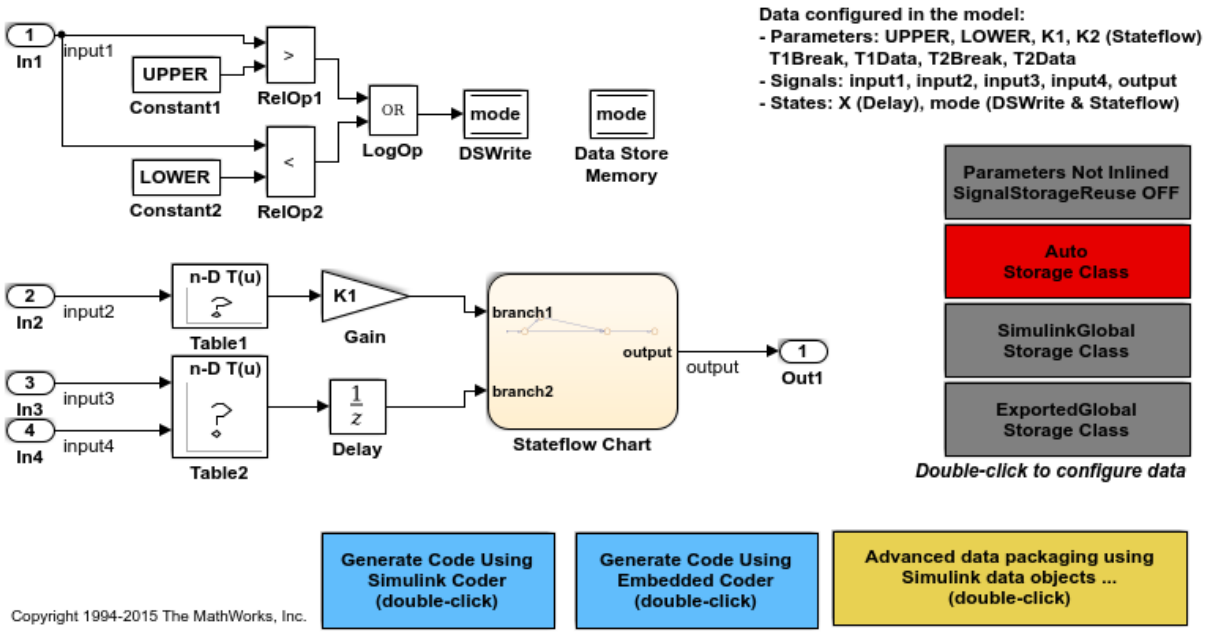
Eliminate Unnecessary Typecasts and Shifts by Matching Data Types

These examples show how to generate efficient code by configuring a block parameter to use the same data type as a signal that the block operates on.

Store Data Type Information in Model

Open the example model `rtwdemo_basicsc`.

```
rtwdemo_basicsc
```



Open the `Gain` block dialog box. The input signal of this block uses the data type `single`.

View the **Parameter Attributes** tab. **Parameter data type** is set to `Inherit: Same as input`. The **Gain** parameter of this block uses the same data type as the input signal.

At the command prompt, convert the numeric variable `K1` to a `Simulink.Parameter` object.

```
K1 = Simulink.Parameter(2);
```

Configure `K1` to appear in the generated code as a global variable by applying the storage class `ExportedGlobal`.

```
K1.CoderInfo.StorageClass = 'ExportedGlobal';
```

On the **Main** tab, click the action button next to the value of the **Gain** parameter. Select **Open Variable**. The value of the **Data type** property is `auto`, which means the parameter object acquires its data type from the block parameters that use the object.

Generate code from the model.

```
rtwbuild('rtwdemo_basicsc')
```

```
### Starting build procedure for model: rtwdemo_basicsc
### Successful completion of build procedure for model: rtwdemo_basicsc
```

The generated file `rtwdemo_basicsc.c` defines the global variable `K1` by using the data type `real32_T`, which corresponds to the data type `single` in Simulink.

```
file = fullfile('rtwdemo_basicsc_grt_rtw','rtwdemo_basicsc.c');
rtwdemodbtype(file,'/* Exported block parameters */','real32_T K1 = 2.0F;',1,1)
```

```
/* Exported block parameters */
real32_T K1 = 2.0F;                    /* Variable: K1
```

The generated code algorithm in the model `step` function uses `K1` directly without typecasting.

```
rtwdemodbtype(file,' rtwdemo_basicsc_DW.X = K1',...
```

```
' rtCP_Table1_bpO1Data, rtCP_Table1_tableData,',1,1)


rtwdemo_basicsc_DW.X = K1 * look1_iflf_binlx(rtwdemo_basicsc_U.input2,
  rtCP_Table1_bpO1Data, rtCP_Table1_tableData, 10U);
```

In the **Gain** block dialog box, on the **Parameter Attributes** tab, you can optionally set **Parameter data type** to `Inherit: Inherit via internal rule` (the default). In this case, the block parameter chooses the same data type as the input signal (`single`). However, when you use `Inherit: Inherit via internal rule`, under other circumstances (for example, when you use fixed-point data types) the block parameter might choose a different data type.

Store Data Type Information in Parameter Object

When you use a `Simulink.Parameter` object to export or import parameter data from the generated code to your custom code, for example by applying the storage class `ImportedExtern`, you can specify data type information in the parameter object. To match the data type of the parameter object with a signal data type, create a `Simulink.NumericType` or `Simulink.AliasType` object. You can strictly control the data type that the parameter object uses in the generated code, eliminating the risk that Simulink chooses a different data type when you make changes to the model.

At the command prompt, create a `Simulink.NumericType` object that represents the data type `single`.

```
myType = Simulink.NumericType;
myType.DataTypeMode = 'Single';
```

Use this data type object as the data type of the parameter object.

```
K1.DataType = 'myType';
```

Use the data type object to set the output data type of the Inport block named `In2`. Due to data type propagation, the input signal of the Gain block also uses this data type.

```
set_param('rtwdemo_basicsc/In2','OutDataTypeStr','myType')
```

In the Gain block dialog box, on the **Parameter Attributes** tab, set **Parameter data type** to `Inherit: Inherit from 'Gain'`.

```
set_param('rtwdemo_basicsc/Gain','ParamDataTypeStr','Inherit: Inherit from ''Gain''')
```

Generate code from the model.

```
rtwbuild('rtwdemo_basicsc')

### Starting build procedure for model: rtwdemo_basicsc
### Successful completion of build procedure for model: rtwdemo_basicsc
```

The global variable `K1` continues to use the data type `real32_T`.

```
file = fullfile('rtwdemo_basicsc_grt_rtw','rtwdemo_basicsc.c');
rtwdemodbtype(file,'/* Exported block parameters */','real32_T K1 = 2.0F;',1,1)


/* Exported block parameters */
real32_T K1 = 2.0F;                    /* Variable: K1
```

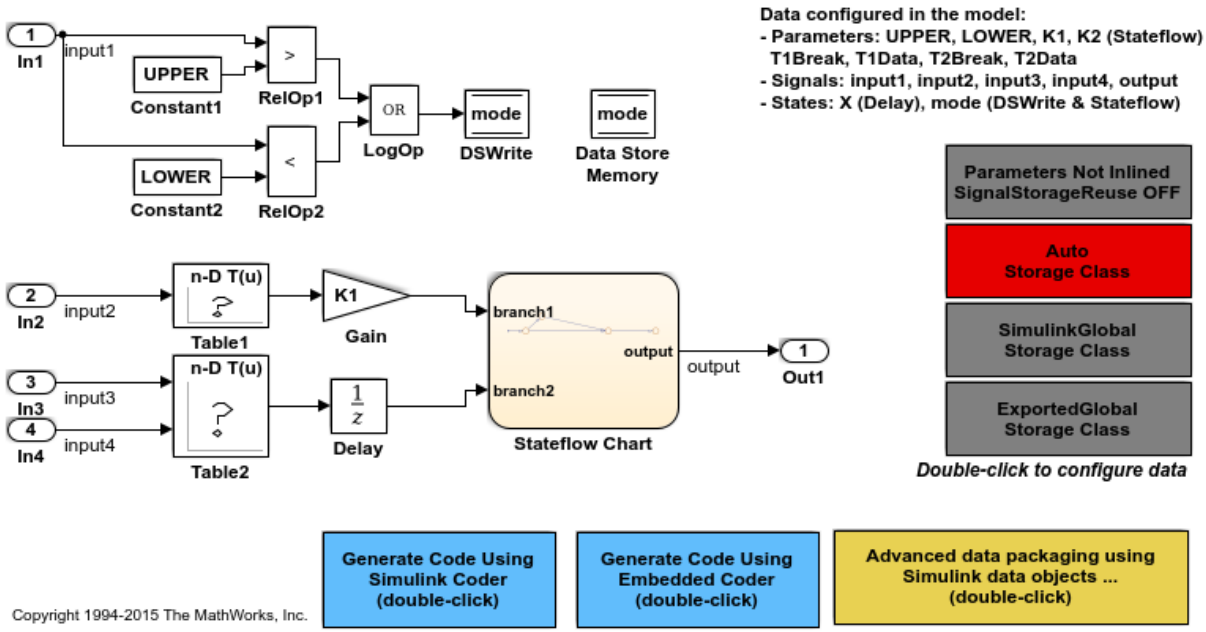
Reduce Memory Consumption by Storing Parameter Value in Small Data Type

When you use a parameter object (for example, `Simulink.Parameter`) to set block parameter values, you can configure the object to appear in the generated code as a tunable global variable. By default, the parameter object and the corresponding global variable typically uses the same data type as the signal or signals on which the block operates. For example, if the input signal of a Gain block uses the data type `int16`, the parameter object typically use the same data type. To reduce the amount of memory that this variable consumes, specify that the variable use a smaller integer data type such as `int8`.

Store Parameter Value in Integer Data Type

Open the example model `rtwdemo_basicsc`.

```
rtwdemo_basicsc
```



In the model, select **Display > Signals and Ports > Port Data Types**. Many of the signals in the model use the data type `single`.

Open the Gain block dialog box. The block uses the MATLAB variable `K1`, which is in the base workspace, to set the value of the **Gain** parameter. The value of the variable is `2`.

At the command prompt, convert `K1` into a `Simulink.Parameter` object. Configure the object to appear in the generated code as a global variable by applying the storage class `ExportedGlobal`.

```
K1 = Simulink.Parameter(2);
K1.CoderInfo.StorageClass = 'ExportedGlobal';
```

Configure the parameter object to use the data type `int8`.

```
K1.DataType = 'int8';
```

In the Gain block dialog box, on the **Parameter Attributes** tab, set **Parameter data type** to `Inherit: Inherit from 'Gain'`. With this setting, the **Gain** parameter of the block inherits the `int8` data type from the parameter object.

```
set_param('rtwdemo_basicsc/Gain','ParamDataTypeStr',...
    'Inherit: Inherit from ''Gain''')
```

Generate code from the model.

```
rtwbuild('rtwdemo_basicsc')
```

```
### Starting build procedure for model: rtwdemo_basicsc
### Successful completion of build procedure for model: rtwdemo_basicsc
```

The generated file `rtwdemo_basicsc.c` defines the global variable `K1` by using the data type `int8_T`, which corresponds to the data type `int8` in Simulink.

```
file = fullfile('rtwdemo_basicsc_grt_rtw','rtwdemo_basicsc.c');
rtwdemodbtype(file,'/* Exported block parameters */','int8_T K1 = 2;',1,1)
```

```
/* Exported block parameters */
int8_T K1 = 2;                         /* Variable: K1
```

The code algorithm in the model `step` function uses `K1` to calculate the output of the Gain block. The algorithm casts `K1` to the data type `real32_T` (`single`) because the signals involved in the calculation use the data type `real32_T`.

```
rtwdemodbtype(file,' rtwdemo_basicsc_DW.X =','1OU);',1,1)
```

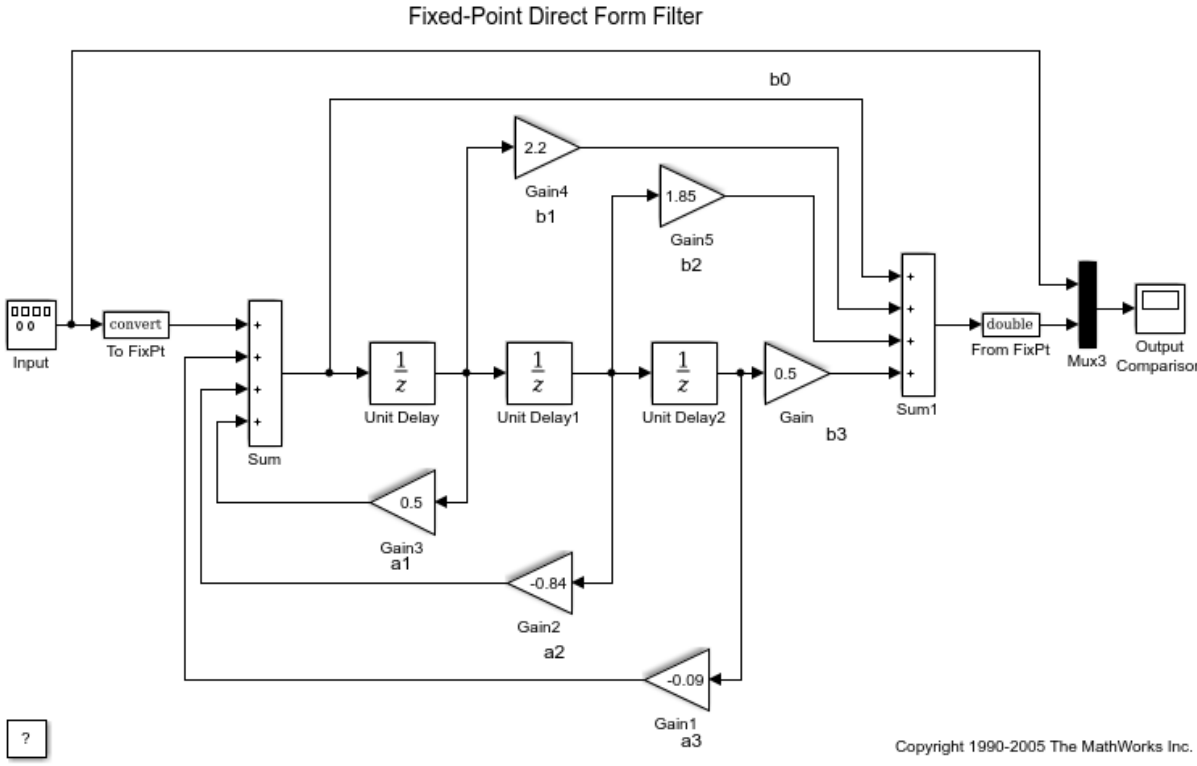
```
    rtwdemo_basicsc_DW.X = (real32_T)K1 * look1_iflf_binlx
      (rtwdemo_basicsc_U.input2, rtCP_Table1_bpOlData, rtCP_Table1_tableData,
       1OU);
```

Store Fixed-Point Parameter Value in Smaller Integer Data Type

Suppose you configure the signals in your model to use fixed-point data types. You want a gain parameter to appear in the generated code as a tunable global variable. You know the range of real-world values that you expect the parameter to assume (for example, between 0 and 4). If you can meet your application requirements despite reduced precision, you can reduce memory consumption by configuring the parameter to use a different data type than the input and output signals of the block.

Open the example model `fxpdemo_direct_form2`.

```
fxpdemo_direct_form2
```



Update the model diagram. Signals in this model use signed fixed-point data types with a word length of 16 and binary-point-only scaling.

Open the Gain5 block dialog box. The **Gain** parameter is set to `1.85`. Suppose you want to configure this parameter.

Set **Gain** to `myGainParam` and click **Apply**.

Click the action button next to the parameter value. Select **Create Variable**.

In the Create New Data dialog box, set **Value** to `Simulink.Parameter(1.85)` and click **Create**. The `Simulink.Parameter` object `myGainParam` appears in the base workspace.

In the `myGainParam` property dialog box, set **Storage class** to `ExportedGlobal` and click **OK**. With this setting, `myGainParam` appears in the generated code as a global variable.

In the block dialog box, on the **Parameter Attributes** tab, set **Parameter minimum** to `0` and **Parameter maximum** to `4`.

Set **Parameter data type** to `fixdt(0,8)` and click **Apply**.

Click the **Show Data Type Assistant** button. The Data Type Assistant shows that the expression `fixdt(0,8)` specifies an unsigned fixed-point data type with a word length of 8 and best-precision scaling. When you simulate or generate code, the block parameter chooses a fraction length (scaling) that enables the data type to represent values between the parameter minimum and maximum (0 and 4) with the best possible precision.

In the Data Type Assistant, set **Scaling** to `Binary point`. Click **Calculate Best-Precision Scaling**, **Fixed-point details**, and **Refresh Details**. The information under **Fixed-point details** shows that a fraction length of 5 can represent the parameter values with a precision of 0.03125.

Set **Scaling** back to `Best precision` and click **OK**. In this example, when you simulate or generate code, the block parameter chooses a fraction length of 5.

You can use these commands at the command prompt to create the object and configure the block:

```
myGainParam = Simulink.Parameter(1.85);
myGainParam.CoderInfo.StorageClass = 'ExportedGlobal';
set_param('fxpdemo_direct_form2/Gain5','Gain','myGainParam')
set_param('fxpdemo_direct_form2/Gain5','ParamMin','0','ParamMax','4')
set_param('fxpdemo_direct_form2/Gain5','ParamDataTypeStr','fixdt(0,8)')
```

Configure the model to produce a code generation report. To reduce clutter in the Command Window, clear the configuration parameter **Verbose build**.

```
set_param('fxpdemo_direct_form2','GenerateReport','on',...
    'LaunchReport','on','RTWVerbose','off')
```

Generate code from the model.

```
rtwbuild('fxpdemo_direct_form2')
```

```
### Starting build procedure for model: fxpdemo_direct_form2
```

```
Code generation utilizes device specific information (e.g., microprocessor word sizes)


### Successful completion of build procedure for model: fxpdemo_direct_form2
```

The generated file `fxpdemo_direct_form2.c` defines the global variable `myGainParam` by using the data type `uint8_T`, which corresponds to the specified word length, 8. The code initializes the variable by using an integer value that, given the fraction length of 5, represents the real-world parameter value 1.85.

```
file = fullfile('fxpdemo_direct_form2_grt_rtw','fxpdemo_direct_form2.c');
rtwdemodbtype(file,'/* Exported block parameters */','uint8_T myGainParam = 59U;',1,1)
```

```
/* Exported block parameters */
uint8_T myGainParam = 59U;             /* Variable: myGainParam
```

The code algorithm uses `myGainParam` to calculate the output of the Gain5 block. The algorithm uses a C shift to scale the result of the calculation.

```
rtwdemodbtype(file,'/* Gain: ''<Root>/Gain5'' */',...
'/* Gain: ''<Root>/Gain'' incorporates:',1,0)
```

```
  /* Gain: '<Root>/Gain5' */
  fxpdemo_direct_form2_B.Gain5 = (int16_T)(myGainParam *
    fxpdemo_direct_form2_B.UnitDelay1 >> 5);
```

Related Examples

- “Optimization Tools and Techniques” on page 53-7
- “Parameter Data Types in the Generated Code” on page 19-79
- “Reuse Parameter Data in Different Data Type Contexts” on page 19-93

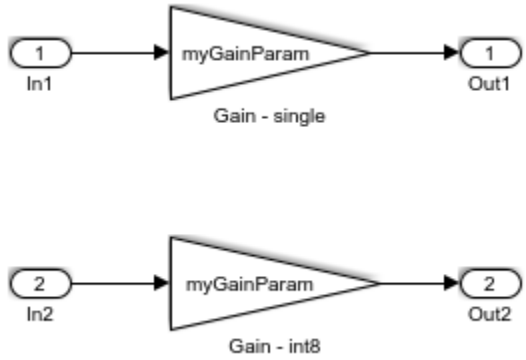
Reuse Parameter Data in Different Data Type Contexts

When you use a `Simulink.Parameter` object or a numeric MATLAB variable to set two or more block parameter values, if the block parameters have different data types, you must explicitly specify the data type of the object or variable. For example, you cannot leave the data type of the parameter object at the default value, `auto`.

Create and Configure Example Model

Create the model `ex_paramdt_contexts`.

```
ex_paramdt_contexts
```



Set these block parameter values:

- `In1` block: **Data type**: `single`
- `In2` block: **Data type**: `int8`

In the block labeled `Gain - single`, set these parameter values:

- **Gain**: `myGainParam`
- **Output data type**: `Inherit: Same as input`
- **Parameter data type**: `Inherit: Same as input`

In the block labeled `Gain - int8`, set these parameter values:

- **Gain**: `myGainParam`
- **Output data type**: `Inherit: Same as input`

- **Parameter data type**: `Inherit: Same as input`

You can use these commands to set the parameter values:

```
set_param('ex_paramdt_contexts/In1','OutDataTypeStr','single')
set_param('ex_paramdt_contexts/In2','OutDataTypeStr','int8')
set_param('ex_paramdt_contexts/Gain - single','Gain','myGainParam',...
    'OutDataTypeStr','Inherit: Same as input',...
    'ParamDataTypeStr','Inherit: Same as input')
set_param('ex_paramdt_contexts/Gain - int8','Gain','myGainParam',...
    'OutDataTypeStr','Inherit: Same as input',...
    'ParamDataTypeStr','Inherit: Same as input')
```

Create a `Simulink.Parameter` object named `myGainParam` in the base workspace. Configure the object to appear in the generated code as a global variable by applying the storage class `ExportedGlobal`. Configure the object to use the data type `int8`.

```
myGainParam = Simulink.Parameter(3);
myGainParam.CoderInfo.StorageClass = 'ExportedGlobal';
myGainParam.DataType = 'int8';
```

In this model, you use the parameter object `myGainParam` to set two block parameter values. The block parameters inherit different data types from the input signals (`single` or `int8`). To use `myGainParam` in these different data type contexts, explicitly specify the data type of the parameter object by setting the `DataType` property to `int8`.

Match Parameter Object Data Type with Signal Data Type

Optionally, use a `Simulink.NumericType` or `Simulink.AliasType` object to set the parameter object data type and one of the signal data types. This technique eliminates unnecessary typecasts and shifts in the generated code due to a mismatch between the parameter object data type and the signal data type.

Create a `Simulink.NumericType` object to represent the data type `int8`.

```
sharedType_int8 = fixdt('int8');
```

Use this data type object to set:

- The data type of the parameter object. Set the `DataType` property to `sharedType_int8`.
- The data type of the `int8` signal. Use the **Data type** parameter in the Inport block dialog box.

```
myGainParam.DataType = 'sharedType_int8';
set_param('ex_paramdt_contexts/In2','OutDataTypeStr','sharedType_int8')
```

The parameter object and the signal use the data type `int8`. To change this data type, adjust the properties of the data type object `sharedType_int8`.

Generate and Inspect Code

Generate code from the model.

```
rtwbuild('ex_paramdt_contexts')
```

```
### Starting build procedure for model: ex_paramdt_contexts
### Successful completion of build procedure for model: ex_paramdt_contexts
```

The generated file `ex_paramdt_contexts.c` defines the global variable `myGainParam` by using the data type `int8_T`, which corresponds to the data type `int8` in Simulink.

```
file = fullfile('ex_paramdt_contexts_grt_rtw','ex_paramdt_contexts.c');
rtwdemodbtype(file,'/* Exported block parameters */','int8_T myGainParam = 3;',1,1)
```

```
/* Exported block parameters */
int8_T myGainParam = 3;                /* Variable: myGainParam
```

The generated code algorithm in the model `step` function uses `myGainParam` to calculate the outputs of the two Gain blocks. In the case of the Gain block whose input signal uses the data type `single`, the code algorithm casts `myGainParam` to the data type `real32_T`, which corresponds to the data type `single` in Simulink.

```
rtwdemodbtype(file,'/* Model step function */',...
    '/* Model initialize function */',1,0)
```

```
/* Model step function */
void ex_paramdt_contexts_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Gain: '<Root>/Gain - single'
   *  Inport: '<Root>/In1'
   */
  ex_paramdt_contexts_Y.Out1 = (real32_T)myGainParam * ex_paramdt_contexts_U.In1;

  /* Outport: '<Root>/Out2' incorporates:
   *  Gain: '<Root>/Gain - int8'
```

```
   *  Inport: '<Root>/In2'
   */
  ex_paramdt_contexts_Y.Out2 = (int8_T)(myGainParam * ex_paramdt_contexts_U.In2);
}
```

Related Examples

- “Parameter Data Types in the Generated Code” on page 19-79
- “Generate Efficient Code by Specifying Data Types for Block Parameters” on page 19-84
- “Block Parameter Representation in the Generated Code” on page 19-47

Organize Block Parameter Values into Structures in the Generated Code

In C code, you use structures to store data in contiguous locations in memory. With structures, you can organize data in the code by using meaningful names. Each structure acts as a namespace, so you can reuse a name to represent multiple data items. Similar to arrays, structures enable you to write code that efficiently transfers and operates on large amounts of data by using pointers.

When you configure the generated code to store block parameter values in memory, for example by applying storage classes to `Simulink.Parameter` objects or by creating model arguments, you can organize the parameter data into structures in the generated code.

- Create a parameter structure in Simulink. Use each field of the structure to set a block parameter value. Optionally, use a `Simulink.Bus` object as the data type of the structure. The generated code creates a structure type definition and a structure variable. Use this technique to finely control the data types and order of the fields.
- If you have an Embedded Coder license, apply a custom storage class, such as the built-in custom storage class `Struct`, to multiple `Simulink.Parameter` objects. Use this technique to avoid the effort of creating the structure in Simulink and to organize both signal and parameter data into a single structure. See "Organize Parameter Data into a Structure by Using the `Struct` Custom Storage Class" on page 23-8.

  When you use this technique, to create a nested structure instead of a flat structure, you must create your own custom storage class by writing TLC code. Consider creating a parameter structure in Simulink instead.

Creating Tunable Parameter Structures

When you create a structure in Simulink, you can configure the generated code to define and initialize a corresponding structure variable or to reuse a structure variable from your existing C code. Use this technique to share a structure of parameter data with your existing code or to generate parameter data whose values you can change (tune) during code execution.

1. Store the structure in a Simulink.Parameter object.
2. Apply a storage class other than `Auto` to the parameter object.

3 Use the fields of the structure to set block parameter values in your model.

The numeric fields of the structure are tunable in the generated code. However, if any field contains a nontunable entity, such as a multidimensional array, none of the structure fields are tunable.

You cannot declare individual substructures or fields within a parameter structure as tunable. You cannot use a `Simulink.Parameter` object as the value of a structure field. Instead, you must store the entire structure in the parameter object.

The fields of parameter structures do not support context-sensitive data typing. However, to match the data type of a field with the data type of another data item in a model, you can use a bus object and a data type object.

1 Use a `Simulink.Bus` object as the data type of the structure.
2 Use a `Simulink.AliasType` or `Simulink.NumericType` object as the data type of the element in the bus object and as the data type of the target data item.

Control Name and File Placement of Structure Type Definition

When you generate code from a model that uses a tunable parameter structure, by default the code generator creates a `struct` type definition that has a generated name such as `struct_z98c0D2qc4btL`. You can control this type name to improve code readability and integrate the generated code with your own code. Create a `Simulink.Bus` object and use it as the data type of the `Simulink.Parameter` object that stores the structure.

To control the file placement of the generated `struct` type definition, use the `HeaderFile` and `DataScope` properties of the bus object. Use this technique to:

- Export the type definition to a header file that you can include in your custom code.
- Import the type definition from your custom code.

  To generate a bus object that represents a `struct` type defined by your code, use the `Simulink.importExternalCTypes` function.

Structures of Parameters

Create a structure in the generated code. The structure stores parameter data.

C Construct

```
typedef struct {
  double G1;
  double G2;
} myStructType;

myStructType myStruct = {
  2.0,
  -2.0
} ;
```

Procedure

At the command prompt, create a structure named `myStruct` with two fields.

```
myStruct.G1 = 2;
myStruct.G2 = -2;
```

Store the structure in a `Simulink.Parameter` object.

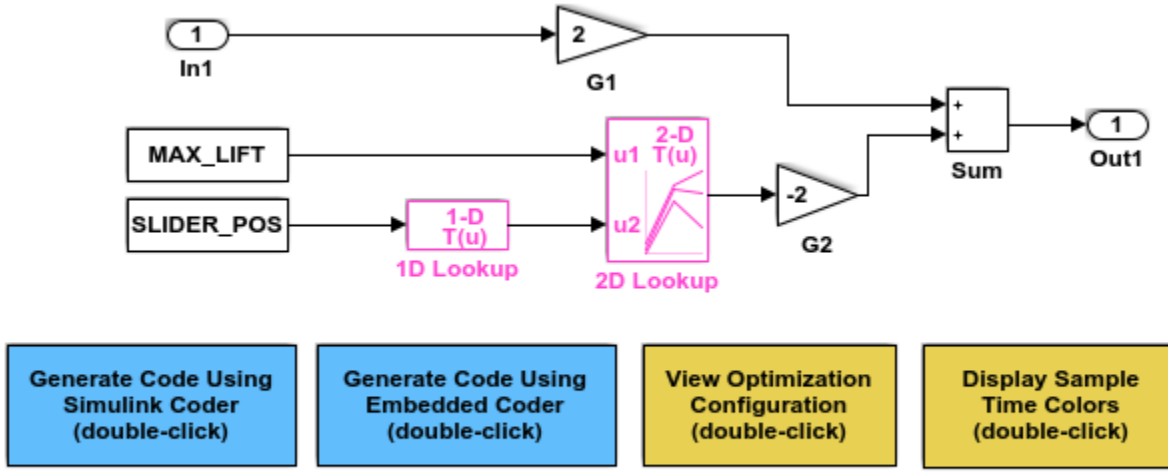
```
myStruct = Simulink.Parameter(myStruct);
```

Apply the storage class `ExportedGlobal` so that the structure appears in the generated code as a global variable.

```
myStruct.CoderInfo.StorageClass = 'ExportedGlobal';
```

Open the example model `rtwdemo_paraminline`.

```
rtwdemo_paraminline
```



In the G1 block dialog box, set **Gain** to `myStruct.G1`.

```
set_param('rtwdemo_paraminline/G1','Gain','myStruct.G1')
```

In the G2 block dialog box, set **Gain** to `myStruct.G2`.

```
set_param('rtwdemo_paraminline/G2','Gain','myStruct.G2')
```

Results

Generate code from the model.

```
rtwbuild('rtwdemo_paraminline')
```

```
### Starting build procedure for model: rtwdemo_paraminline
### Successful completion of build procedure for model: rtwdemo_paraminline
```

The generated header file `rtwdemo_paraminline_types.h` defines a structure type with a randomized name.

```
file = fullfile('rtwdemo_paraminline_grt_rtw',...
    'rtwdemo_paraminline_types.h');
```

```
rtwdemodbtype(file,'typedef struct {','} struct_6h72eH5WFuEIyQr5YrdGuB;',...
    1,1)
```

```
typedef struct {
  real_T G1;
  real_T G2;
} struct_6h72eH5WFuEIyQr5YrdGuB;
```

The source file `rtwdemo_paraminline.c` defines and initializes the structure variable `myStruct`.

```
file = fullfile('rtwdemo_paraminline_grt_rtw','rtwdemo_paraminline.c');
rtwdemodbtype(file,'struct_6h72eH5WFuEIyQr5YrdGuB myStruct',...
    '/* Variable: myStruct',1,1)
```

```
struct_6h72eH5WFuEIyQr5YrdGuB myStruct = {
  2.0,
  -2.0
} ;                                    /* Variable: myStruct
```

Specify Name of Structure Type

Optionally, specify a name to use for the structure type definition (`struct`).

Create a `Simulink.Bus` object that represents the structure type.

```
Simulink.Bus.createObject(myStruct.Value);
```

The default name of the object is `slBus1`. Change the name by copying the object into a new MATLAB variable.

```
myStructType = slBus1.copy;
```

Use the bus object as the data type of the parameter object.

```
myStruct.DataType = 'Bus: myStructType';
```

Generate code from the model.

```
rtwbuild('rtwdemo_paraminline')
```

```
### Starting build procedure for model: rtwdemo_paraminline
### Successful completion of build procedure for model: rtwdemo_paraminline
```

The code generates the definition of the structure type `myStructType` and uses this type to define the global variable `myStruct`.

```
rtwdemodbtype(file,'myStructType myStruct = {','/* Variable: myStruct',...
    1,1)
```

```
myStructType myStruct = {
  2.0,
  -2.0
} ;                                     /* Variable: myStruct
```

Structure Padding

By default, the code generator does not explicitly add padding fields to structure types. Structure types can appear in the generated code through, for example, the default data structures (see "Default Data Structures in the Generated Code" (Simulink Coder)), `Simulink.Bus` objects, and parameter structures that you use in a model.

However, when you use a code replacement library with Embedded Coder, you can specify data alignment (including structure padding) as part of the replacement library. For more information, see "Provide Data Alignment Specifications for Compilers" on page 51-135.

Related Examples

- "Migration to Structure Parameters" (Simulink)
- "Create Tunable Calibration Parameter in the Generated Code" on page 19-60
- "Block Parameter Representation in the Generated Code" on page 19-47
- "Specify Instance-Specific Parameter Values for Reusable Referenced Model" on page 19-65
- "Organize Related Block Parameter Definitions in Structures" (Simulink)
- "Access Structured Data Through a Pointer That External Code Defines" on page 23-27
- "Exchange Structured and Enumerated Data Between Generated and External Code" on page 21-28

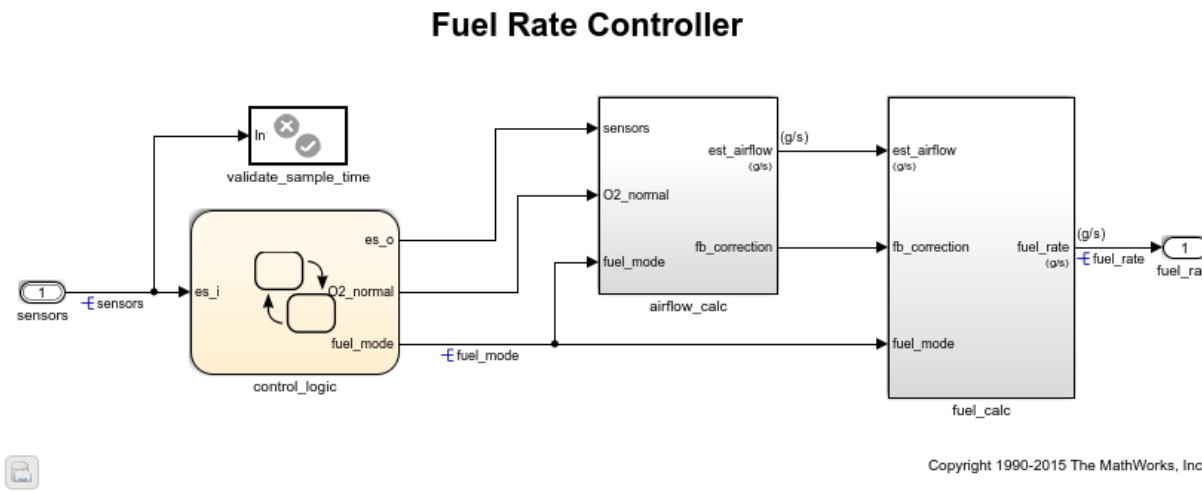
Switch Between Sets of Parameter Values During Simulation and Code Execution

To store multiple independent sets of values for the same block parameters, you can use an array of structures. To switch between the parameter sets, create a variable that acts as an index into the array, and change the value of the variable. You can change the value of the variable during simulation and, if the variable is tunable, during execution of the generated code.

Explore Example Model

Open this example model:

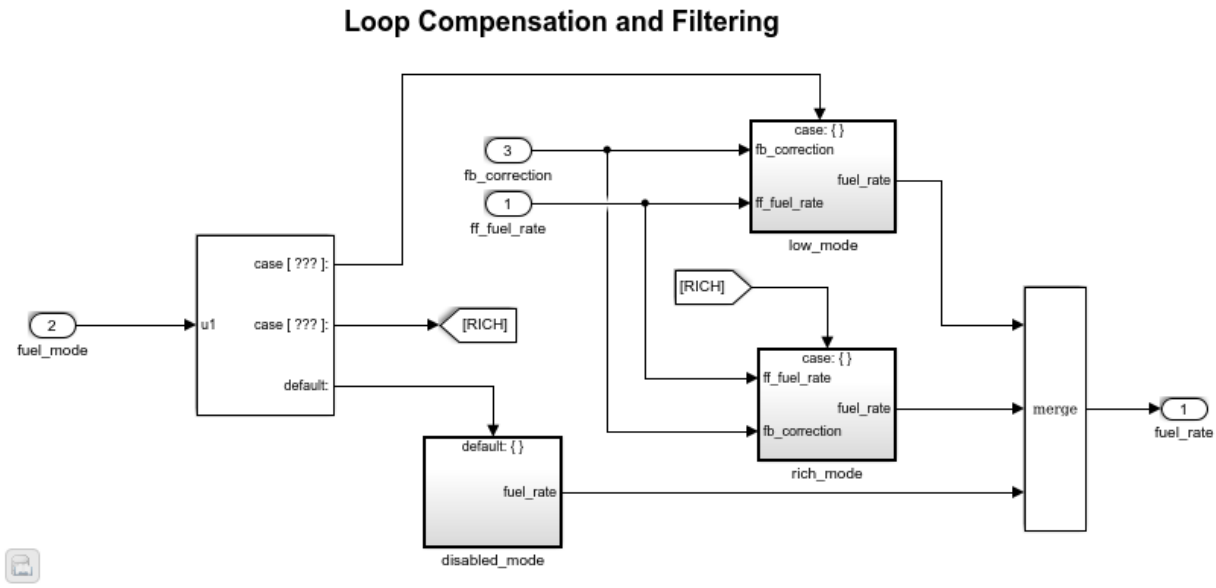
```
open_system('sldemo_fuelsys_dd_controller')
```



This model represents the fueling system of a gasoline engine. The output of the model is the rate of fuel flow to the engine.

Navigate to the `switchable_compensation` nested subsystem.

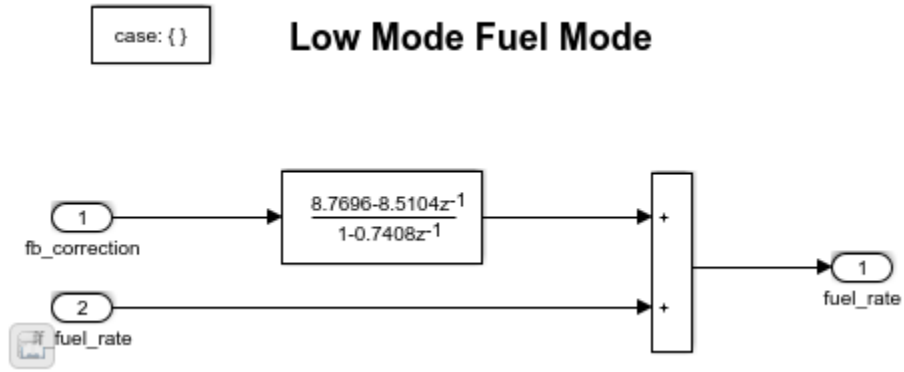
```
open_system(['sldemo_fuelsys_dd_controller/fuel_calc/',...
     'switchable_compensation'])
```



This subsystem corrects and filters noise out of the fuel rate signal. The subsystem uses different filter coefficients based on the fueling mode, which the control logic changes based on sensor failures in the engine. For example, the control algorithm activates the `low_mode` subsystem during normal operation. It activates the `rich_mode` subsystem in response to sensor failure.

Open the `low_mode` subsystem.

```
open_system(['sldemo_fuelsys_dd_controller/fuel_calc/',...
     'switchable_compensation/low_mode'])
```



The Discrete Filter block filters the fuel rate signal. In the block dialog box, the **Numerator** parameter sets the numerator coefficients of the filter.

The sibling subsystem `rich_mode` also contains a Discrete Filter block, which uses different coefficients.

Update the model diagram to display the signal data types. The input and output signals of the block use the single-precision, floating-point data type `single`.

In the lower-left corner of the model, click the data dictionary badge. The data dictionary for this model, `sldemo_fuelsys_dd_controller.sldd`, opens in the Model Explorer.

In the **Contents** pane, view the properties of the `Simulink.NumericType` objects, such as `s16En15`. All of these objects currently represent the single-precision, floating-point data type `single`. The model uses these objects to set signal data types, including the input and output signals of the Discrete Filter blocks.

Suppose that during simulation and execution of the generated code, you want each of these subsystems to switch between different numerator coefficients based on a variable whose value you control.

Store Parameter Values in Array of Structures

Store the existing set of numerator coefficients in a `Simulink.Parameter` object whose value is a structure. Each field of the structure stores the coefficients for one of the Discrete Filter blocks.

```
lowBlock = ['sldemo_fuelsys_dd_controller/fuel_calc/'...
    'switchable_compensation/low_mode/Discrete Filter'];
```

```
richBlock = ['sldemo_fuelsys_dd_controller/fuel_calc/'...
    'switchable_compensation/rich_mode/Discrete Filter'];
params.lowNumerator = eval(get_param(lowBlock,'Numerator'));
params.richNumerator = eval(get_param(richBlock,'Numerator'));
params = Simulink.Parameter(params);
```

Copy the value of `params` into a temporary variable. Modify the field values in this temporary structure, and assign the modified structure as the second element of `params`.

```
temp = params.Value;
temp.lowNumerator = params.Value.lowNumerator * 2;
temp.richNumerator = params.Value.richNumerator * 2;
params.Value(2) = temp;
clear temp
```

The value of `params` is an array of two structures. Each structure stores one set of filter coefficients.

Create Variable to Switch Between Parameter Sets

Create a `Simulink.Parameter` object named `Ctrl`.

```
Ctrl = Simulink.Parameter(2);
Ctrl.DataType = 'uint8';
```

In the `low_mode` subsystem, in the Discrete Filter block dialog box, set the **Numerator** parameter to the expression `params(Ctrl).lowNumerator`.

```
set_param(lowBlock,'Numerator','params(Ctrl).lowNumerator');
```

In the Discrete Filter block in the `rich_mode` subsystem, set the value of the **Numerator** parameter to `params(Ctrl).richNumerator`.

```
set_param(richBlock,'Numerator','params(Ctrl).richNumerator');
```

The expressions select one of the structures in `params` by using the variable `Ctrl`. The expressions then dereference one of the fields in the structure. The field value sets the values of the numerator coefficients.

To switch between the sets of coefficients, you change the value of `Ctrl` to the corresponding index in the array of structures.

Use Bus Object as Data Type of Array of Structures

Optionally, create a `Simulink.Bus` object to use as the data type of the array of structures. You can:

- Control the shape of the structures.
- For each field, control characteristics such as data type and physical units.
- Control the name of the `struct` type in the generated code.

Use the function `Simulink.Bus.createObject` to create the object and rename the object as `paramsType`.

```
Simulink.Bus.createObject(params.Value)
paramsType = slBus1;
clear slBus1
```

You can use the `Simulink.NumericType` objects from the data dictionary to control the data types of the structure fields. In the bus object, use the name of a data type object to set the `DataType` property of each element.

```
paramsType.Elements(1).DataType = 's16En15';
paramsType.Elements(2).DataType = 's16En7';
```

Use the bus object as the data type of the array of structures.

```
params.DataType = 'Bus: paramsType';
```

Use Enumerated Type for Switching Variable

Optionally, use an enumerated type as the data type of the switching variable. You can associate each of the parameter sets with a meaningful name and restrict the allowed values of the switching variable.

Create an enumerated type named `FilterCoeffs`. Create an enumeration member for each of the structures in `params`. Set the underlying integer value of each enumeration member to the corresponding index in `params`.

```
Simulink.defineIntEnumType('FilterCoeffs',{'Weak','Aggressive'},[1 2])
```

Use the enumerated type as the data type of the switching variable. Set the value of the variable to `Aggressive`, which corresponds to the index `2`.

```
Ctrl.Value = FilterCoeffs.Aggressive;
```

Add New Objects to Data Dictionary

Add the objects that you created to the data dictionary `sldemo_fuelsys_dd_controller.sldd`.

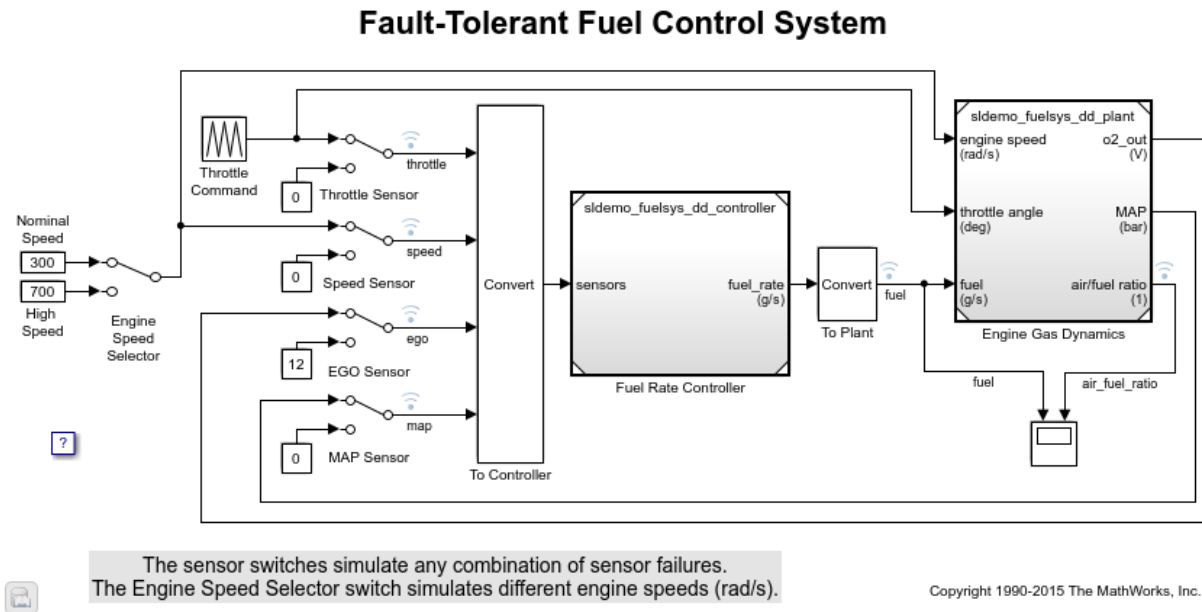
```
dictObj = Simulink.data.dictionary.open('sldemo_fuelsys_dd_controller.sldd');
sectObj = getSection(dictObj,'Design Data');
addEntry(sectObj,'Ctrl',Ctrl)
addEntry(sectObj,'params',params)
addEntry(sectObj,'paramsType',paramsType)
```

You can also store enumerated types in data dictionaries. However, you cannot import the enumerated type in this case because you cannot save changes to `sldemo_fuelsys_dd_controller.sldd`. For more information about storing enumerated types in data dictionaries, see "Enumerations in Data Dictionary" (Simulink).

Switch Between Parameter Sets During Simulation

Open the example model `sldemo_fuelsys_dd`, which references the controller model `sldemo_fuelsys_dd_controller`.

```
open_system('sldemo_fuelsys_dd')
```



Set the simulation stop time to `Inf` so that you can interact with the model during simulation.

Begin a simulation run and open the Scope block dialog box. The scope shows that the fuel flow rate (the `fuel` signal) oscillates with significant amplitude during normal operation of the engine.

In the Model Explorer, view the contents of the data dictionary `sldemo_fuelsys_dd_controller.sldd`. Set the value of `Ctrl` to `FilterCoeffs.Weak`.

Update the `sldemo_fuelsys_dd` model diagram. The scope shows that the amplitude of the fuel rate oscillations decreases due to the less aggressive filter coefficients.

Stop the simulation.

Generate and Inspect Code

If you have Simulink Coder software, you can generate code that enables you to switch between the parameter sets during code execution.

In the Model Explorer, view the contents of the data dictionary `sldemo_fuelsys_dd_controller.sldd`. In the **Contents** pane, set **Column View** to `Storage Class`.

Use the **StorageClass** column to apply the storage class `ExportedGlobal` to `params` so that the array of structures appears as a tunable global variable in the generated code. Apply the same storage class to `Ctrl` so that you can change the value of the switching variable during code execution.

Alternatively, to configure the objects, use these commands:

```
tempEntryObj = getEntry(sectObj,'params');
params = getValue(tempEntryObj);
params.StorageClass = 'ExportedGlobal';
setValue(tempEntryObj,params);

tempEntryObj = getEntry(sectObj,'Ctrl');
Ctrl = getValue(tempEntryObj);
Ctrl.StorageClass = 'ExportedGlobal';
setValue(tempEntryObj,Ctrl);
```

Generate code from the controller model.

```
rtwbuild('sldemo_fuelsys_dd_controller')

### Starting build procedure for model: sldemo_fuelsys_dd_controller
```

```
### Successful completion of code generation for model: sldemo_fuelsys_dd_controller
```

In the code generation report, view the header file `sldemo_fuelsys_dd_controller_types.h`. The code defines the enumerated data type `FilterCoeffs`.

```
file = fullfile('sldemo_fuelsys_dd_controller_ert_rtw',...
    'sldemo_fuelsys_dd_controller_types.h');
rtwdemodbtype(file,'#ifndef DEFINED_TYPEDEF_FOR_FilterCoeffs_',...
    '/* Forward declaration for rtModel */',1,0)
```

```
#ifndef DEFINED_TYPEDEF_FOR_FilterCoeffs_
#define DEFINED_TYPEDEF_FOR_FilterCoeffs_

typedef enum {
  Weak = 1,                            /* Default value */
  Aggressive
} FilterCoeffs;

#endif
```

The code also defines the structure type `paramsType`, which corresponds to the `Simulink.Bus` object. The fields use the single-precision, floating-point data type from the model.

```
rtwdemodbtype(file,'#ifndef DEFINED_TYPEDEF_FOR_paramsType_',...
    '#ifndef DEFINED_TYPEDEF_FOR_FilterCoeffs_',1,0)
```

```
#ifndef DEFINED_TYPEDEF_FOR_paramsType_
#define DEFINED_TYPEDEF_FOR_paramsType_

typedef struct {
  real32_T lowNumerator[2];
  real32_T richNumerator[2];
} paramsType;

#endif
```

View the source file `sldemo_fuelsys_dd_controller.c`. The code uses the enumerated type to define the switching variable `Ctrl`.

```
file = fullfile('sldemo_fuelsys_dd_controller_ert_rtw',...
    'sldemo_fuelsys_dd_controller.c');
```

```
rtwdemodbtype(file,'FilterCoeffs Ctrl = Aggressive;',...
    '/* Block signals (auto storage) */',1,0)


FilterCoeffs Ctrl = Aggressive;        /* Variable: Ctrl
                                        * Referenced by:
                                        *   '<S12>/Discrete Filter'
                                        *   '<S13>/Discrete Filter'
                                        */
```

The code also defines the array of structures `params`.

```
rtwdemodbtype(file,'/* Exported block parameters */',...
    '/* Variable: params',1,1)


/* Exported block parameters */
paramsType params[2] = { {
    { 8.7696F, -8.5104F },

    { 0.0F, 0.2592F }
  }, { { 17.5392F, -17.0208F },

    { 0.0F, 0.5184F }
  } } ;                                /* Variable: params
```

The code algorithm in the model `step` function uses the switching variable to index into the array of structures.

To switch between the parameter sets stored in the array of structures, change the value of `Ctrl` during code execution.

Related Examples

- “Tune and Experiment with Block Parameter Values” (Simulink)
- “Block Parameter Representation in the Generated Code” (Simulink Coder)
- “Organize Related Block Parameter Definitions in Structures” (Simulink)
- “Access Structured Data Through a Pointer That External Code Defines” on page 23-27

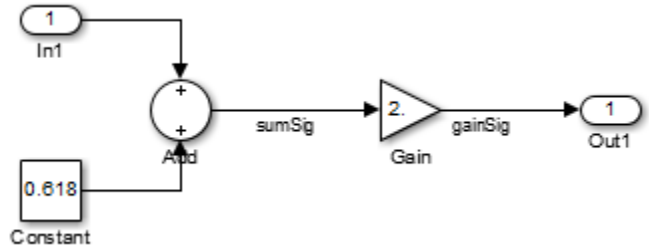
Signal Representation in Generated Code

In this section...

The code generator offers a number of options that let you control how signals in your model are stored and represented in the generated code.

- 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.
- Declare signals as *test points* to store them in unique memory locations
- Reduce memory usage by instructing the code generator to store signals in reusable buffers.
- Control whether or not signals declared in generated code are interfaceable (visible) to externally written code. You can also specify that signals are to be stored in locations declared by externally written code.
- Preserve the symbolic names of signals in generated code by using signal labels.

The discussion in the following sections refers to code generated from `signal_examp`, the model shown in the next figure.



signal_examp Model

Signal Storage Concepts

This section discusses structures and concepts you must understand to choose the best signal storage options for your application:

- The global block I/O data structure *`model`*`_B`
- The concept of signal *storage classes* as used by the code generator

Global Block I/O Structure

By default, the code generator attempts to optimize memory usage by sharing signal memory and using local variables.

However, under a number of circumstances you should place signals in global memory. For example,

- You might want a signal to be stored in a structure that is visible to externally written code.
- The number and/or size of signals in your model might exceed the stack space available for local variables.

In such cases, it is possible to override the default behavior and store selected signals in a model-specific *global block I/O data structure*. The global block I/O structure is called *`model`*`_B` (in earlier versions this was called `rtB`).

The following code shows how *`model`*`_B` is defined and declared in code generated (with signal storage optimizations off) from the `signal_examp` model shown in the signal_examp Model figure.

```
(in signal_examp.h)
/* Block signals (auto storage) */
extern B_signal_examp_T signal_examp_B;

(in signal_examp.c)
/* Block signals (auto storage) */
B_signal_examp_T signal_examp_B;
```

Field names for signals stored in *model*_B are generated according to the rules described in "Symbolic Naming Conventions for Signals" on page 19-118.

In certain cases, the code generator places signals in the block I/O structure, even when you specify the storage class for the signal object as Auto or if you enable the option to reuse the signal. This override occurs when the values of these signals need to be persistent across time step. In such cases, the only way to represent these signals locally in generated code is to change the semantics of your Simulink model.

Signal Storage Class

The *storage class* property of a signal specifies how the product declares and stores the signal. In some cases this specification is qualified by more options.

In the context of the code generator, the term "storage class" is not synonymous with the term *storage class specifier*, as used in the C language.

Default Storage Class

Auto is the default storage class and is the storage class you should use for signals that you do not need to interface to external code. Signals with Auto storage class can be stored in local and/or shared variables or in a global data structure. The form of storage depends on the **Signal storage reuse**, **Reuse local block outputs**, and **Enable local block outputs** options, and on available stack space. See "Signals with Auto Storage Class" on page 19-115 for a full description of code generation options for signals with Auto storage class.

Explicitly Assigned Storage Classes

Signals with storage classes other than Auto are stored either as members of an appropriate global data structure, such as *model*_B or *model*_U, or in unstructured global variables, independent of the global data structures. These storage classes are for signals that you want to monitor and/or interface to external code.

The **Signal storage reuse**, **Enable local block outputs**, **Reuse local block outputs**, and **Eliminate superfluous local variables (expression folding)** optimizations do not apply to signals with storage classes other than `Auto`.

Use the Signal Properties dialog box to assign these storage classes to signals:

- `SimulinkGlobal`: The signal is stored as a field of a global data structure such as the block I/O structure. Signals with `SimulinkGlobal` storage class must have unique signal names. See "Control Signals and States in Code by Applying Storage Classes" on page 19-123 for more information.
- `ExportedGlobal`: The signal is stored in a global variable, independent of the global data structures. *`model`*`.h` exports the variable. Signals with `ExportedGlobal` storage class must have unique signal names. See "Control Signals and States in Code by Applying Storage Classes" on page 19-123 for more information.
- `ImportedExtern`: *`model`*`_private.h` declares the signal as an `extern` variable. Your code must supply the variable definition. Signals with `ImportedExtern` storage class must have unique signal names. See "Control Signals and States in Code by Applying Storage Classes" on page 19-123 for more information.
- `ImportedExternPointer`: *`model`*`_private.h` declares the signal as an `extern` pointer. Your code must define a valid pointer variable. Signals with `ImportedExtern` storage class must have unique signal names. See "Control Signals and States in Code by Applying Storage Classes" on page 19-123 for more information.

Signals with Auto Storage Class

Options are available for signals with `Auto` storage class:

- **Signal storage reuse**
- **Enable local block outputs**
- **Reuse local block outputs**
- **Eliminate superfluous local variables (expression folding)**

Use these options to control signal memory reuse and choose local or global (*`model`*`_B`) storage for signals. These options are on the **All Parameters** tab of the Configuration Parameters dialog box.

These options interact. When the **Signal storage reuse** option is selected,

- The **Reuse local block outputs** option is enabled and selected, and signal memory is reused whenever possible, reducing stack size where signals are being buffered in local variables.
- The **Enable local block outputs** option is enabled and selected. This parameter lets you choose whether reusable signal variables are declared as local variables in functions or as members of *`model`*`_B`.
- The **Eliminate superfluous local variables (expression folding)** is enabled and selected, and block computations collapse into single expressions.

The following code examples illustrate the effects of the **Signal storage reuse**, **Enable local block outputs**, and **Reuse local block outputs** options. The examples were generated from the `signal_examp` model (see figure signal_examp Model). For clarity in showing the individual Gain and Sum block computation, expression folding is off in this example.

This code example shows signal storage optimization, with **Signal storage reuse**, **Enable local block outputs**, and **Reuse local block outputs** selected. The local variable `rtb_gainSig` holds the outputs of the Sum and Gain blocks.

```
/* Model step function */
void signal_examp_step(void)
{
  real_T rtb_gainSig;

  /* Sum: '<Root>/Sum' incorporates:
   *  Constant: '<Root>/Constant'
   *  Inport: '<Root>/In1'
   */
  rtb_gainSig = signal_examp_U.In1 + signal_examp_P.Constant_Value;

  /* Gain: '<Root>/Gain' */
  rtb_gainSig *= signal_examp_P.Gain_Gain;

  /* Outport: '<Root>/Out1' */
  signal_examp_Y.Out1 = rtb_gainSig;
}
```

This example shows the code with **Signal storage reuse** cleared. The global variable `signal_example_B.sumSig` holds the output of the Sum block.

```
/* Model step function */
void signal_examp_step(void)
{
  /* Sum: '<Root>/Sum' incorporates:
   *  Constant: '<Root>/Constant'
   *  Inport: '<Root>/In1'
   */
```

```
  signal_examp_B.sumSig = signal_examp_P.Constant_Value + signal_examp_U.In1;

  /* Outport: '<Root>/Out1' incorporates:
   *  Gain: '<Root>/Gain'
   */
  signal_examp_Y.Out1 = signal_examp_P.Gain_Gain * signal_examp_B.sumSig;
}
```

In large models, disabling **Signal storage reuse** can significantly increase RAM and ROM usage. Therefore, this approach is not recommended for code deployment; however it can be useful in rapid prototyping environments.

The following table summarizes the possible combinations of the **Signal storage reuse** / **Reuse block outputs** and **Enable local block outputs** options.

	Signal storage reuse and Reuse local block outputs ON	Signal storage reuse OFF (Reuse local block outputs disabled)
Enable local block outputs ON	Reuse signals in local memory (fully optimized)	N/A
Enable local block outputs OFF	Reuse signals in *model*_B structure	Individual signal storage in *model*_B structure

Signals with Test Points

A *test point* is a signal that is stored in a unique location that other signals cannot share or reuse. See "Test Points" (Simulink) for information about including test points in your model.

When you generate code for models that include test points, the build process allocates a separate memory buffer for each test point. Test points are stored as members of an appropriate global data structure such as *model*_B or *model*_U.

Declaring a signal as a test point disables the following options for that signal. This can lead to increased code and data size. You do not lose the benefits of optimized storage for other signals in your model.

- **Signal storage reuse**
- **Enable local block outputs**
- **Reuse local block outputs**

- **Eliminate superfluous local variables (expression folding)**

For an example of storage declarations and code generated for a test point, see "Summary of Signal Storage Class Options" on page 19-119.

If you have an Embedded Coder license, you can specify that the build process ignore test points in the model, allowing optimal buffer allocation, using the "Ignore test point signals" (Simulink Coder) parameter. Ignoring test points facilitates transitioning from prototyping to deployment and avoids accidental degradation of generated code due to workflow artifacts. For more information, see "Ignore test point signals" (Simulink Coder).

Symbolic Naming Conventions for Signals

When signals have a storage class other than `Auto`, the code generator preserves symbolic information about the signals or their originating blocks in the generated code.

For labeled signals, field names in *`model`*`_B` derive from the signal names. In the following example, the field names *`model`*`_B.sumSig` and *`model`*`_B.gainSig` are derived from the corresponding labeled signals in the `signal_examp` model (shown in figure signal_examp Model).

```
/* Block signals (auto storage) */
typedef struct _BlockIO_signal_examp {
    real_T sumSig;                       /* '<Root>/Add' */
    real_T gainSig;                      /* '<Root>/Gain' */
} BlockIO_signal_examp;
```

When you clear the **Signal storage reuse** optimization, `sumSig` is not part of *`model`*`_B`, and a local variable is used for it instead. For unlabeled signals, *`model`*`_B` field names are derived from the name of the source block or subsystem.

The components of a generated signal label are

- The root model name, followed by
- The name of the generating signal object, followed by
- Unique *name-mangling* text (if required)

The number of characters that a signal label can have is limited by the **Maximum identifier length** parameter specified on the **Symbols** pane of the Configuration

Parameters dialog box. See "Construction of Generated Identifiers" (Simulink Coder) for more detail.

When a signal has `Auto` storage class, the build process controls generation of variable or field names without regard to signal labels.

Summary of Signal Storage Class Options

The next table shows, for each signal storage class option, the variable declaration and the code generated for Sum (`sumSig`) and Gain (`gainSig`) block outputs of the model shown in figure signal_examp Model.

Storage Class	Declaration	Code

`Auto`
 	
(with **Signal storage reuse** optimizations on)
	In *`model`*`.c` or *`model`*`.cpp`
	
	`real_T rtb_sumSig;`
		`rtb_sumSig = signal_examp_U.In1 +`
		`  signal_examp_P.Constant_Value;`
		`rtb_sumSig *=`
		`  signal_examp_P.Gain_Gain;`
		`signal_examp_Y.Out1 = rtb_sumSig;`

Test point (for `sumSig` only)	In *`model`*`.h`
	
	`typedef struct`
	`_BlockIO_signal_examp`
	`{`
	`  real_T sumSig;`
	`}`
	`BlockIO_signal_examp;`
	
	In *`model`*`.c` or *`model`*`.cpp`
	
	`BlockIO_signal_examp`
	`signal_examp_B;`
	`real_T rtb_gainSig;`
		`signal_examp_B.sumSig =`
		`  signal_examp_U.In1 +`
		`  signal_examp_P.Constant_Value;`
		`rtb_gainSig =`
		`  signal_examp_B.sumSig *`
		`  signal_examp_P.Gain_Gain;`
		`signal_examp_Y.Out1 = rtb_gainSig;`

`ExportedGlobal` (for `sumSig` only)	In *`model`*`.h`
	
	`extern real_T sumSig;`
	
	In *`model`*`.c` or *`model`*`.cpp`
	
	`real_T sumSig;`
	`real_T rtb_gainSig;`
		`sumSig = signal_examp_U.In1 +`
		`signal_examp_P.Constant_Value;`
		`rtb_gainSig = sumSig *`
		`  signal_examp_P.Gain_Gain;`
		`signal_examp_Y.Out1 = rtb_gainSig;`

`ImportedExtern`	In *`model`*`_private.h`
	
	`extern real_T sumSig;`
	
	In *`model`*`.c` or *`model`*`.cpp`
	
	`real_T rtb_gainSig;`
		`sumSig = signal_examp_U.In1 +`
		`signal_examp_P.Constant_Value;`
		`rtb_gainSig = sumSig *`
		`  signal_examp_P.Gain_Gain;`
		`signal_examp_Y.Out1 = rtb_gainSig;`

Storage Class	Declaration	Code

`ImportedExternPointer`	In *`model`*`_private.h`
	`extern real_T *sumSig;`
	In *`model`*`.c` or *`model`*`.cpp`
	`real_T rtb_gainSig;`
		`(*sumSig) = signal_examp_U.In1 + signal_examp_P.Constant_Value;`
		`rtb_gainSig = (*sumSig) * signal_examp_P.Gain_Gain;`
		`signal_examp_Y.Out1 = rtb_gainSig;`

Interfaces for Monitoring Signals

The code generator includes

- Support for developing a Target Language Compiler API for monitoring signals and states independent of external mode. See "Input Signal Functions" (Simulink Coder) and "Output Signal Functions" (Simulink Coder).
- A C application program interface (API) for monitoring signals and states independent of external mode. See "Exchange Data Between Generated and External Code Using C API" (Simulink Coder) for information.
- An interface for exporting ASAP2 files, which you customize to use signal objects. For details, see "Export ASAP2 File for Data Measurement and Calibration" (Simulink Coder).

Share Data Between Code Generated from Simulink, Stateflow, and MATLAB

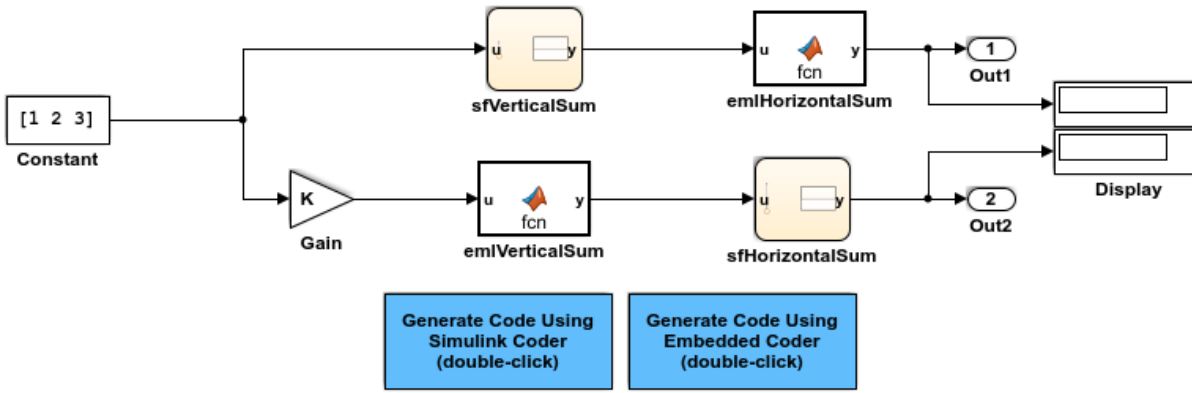
Stateflow and MATLAB Coder can fully define their data definitions, or they can inherit them from Simulink. Data definition capabilities include:

- Inheriting input/output data types and sizes from Simulink.
- Parameterized data types and sizes. That is, data type and size may be specified as a function of another data's type and size, e.g., `type(y)=type(u)` and `size(y)=size(u)`.
- Inferred output size and type from Simulink via signal attribute back propagation.
- Parameter scoped data, which allows referencing Simulink parameters in Stateflow and MATLAB.

Open Example Model

Open the example model `rtwdemo_dynamicio`.

```
open_system('rtwdemo_dynamicio')
```



Instructions

1. Compile the model (**Simulation > Update Diagram**) and note the displayed signal types and sizes.
2. Change the data type and/or size of the Constant block and recompile the model. Note that the attributes of the signals automatically adapt to the Constant block specification.
3. Generate and inspect code using the blue buttons in the model. Note that `K` is shared by the Gain and sfVerticalSum block.

Notes

- The data type and size of all Stateflow and MATLAB data is inherited from Simulink.
- The Gain block and the Stateflow chart sfVerticalSum share the Simulink parameter `K`, which is defined in the MATLAB workspace as a Simulink.Parameter with `SimulinkGlobal` storage class (i.e., `rtP.K` in the generated code).

Related Examples

- “Control Signals and States in Code by Applying Storage Classes” on page 19-123

- “Control Signal and State Initialization in the Generated Code” (Simulink Coder)
- “Maximize Signal Storage Optimization” (Simulink Coder)
- “Default Data Structures in the Generated Code” (Simulink Coder)
- “Storage Classes for Signals Used with Model Blocks” (Simulink Coder)
- “Control Signal Data Types” (Simulink)

Control Signals and States in Code by Applying Storage Classes

Signals and block states appear in the generated code as variables. For basic information about signal representation in the generated code, see "Signal Representation in Generated Code" on page 19-112. For basic information about state representation, see "Discrete Block State Naming in Generated Code" (Simulink Coder) and "Continuous Block State Naming in Generated Code" (Simulink Coder).

In a model, you can assign each signal and state a *storage class* to determine the variable scope in the generated code. You can use storage classes to monitor signal and state data during execution and interface signals and states to externally written code.

To specify storage classes for signals and states, you can use *signal objects*, which you store in a workspace or data dictionary. Signal objects are objects of the class `Simulink.Signal`. If you create your own data class package, signal objects are also objects of the subclass of `Simulink.Signal` that your package defines. For basic information about data objects, see "Data Objects" (Simulink).

You can apply storage classes directly to signal lines and block states by using the Signal Properties dialog box or the **State Attributes** tab of a block dialog box. This technique does not require you to store a signal object in a workspace.

You can interface test points and other signals that are stored as members of *`model`*`_B`, or of another global data structure such as *`model`*`_U`, to your code. To do this, your code must know the address of the global data structure where the data is stored, and other information. This information is not automatically exported. The code generator provides C/C++ and Target Language Compiler APIs that give your code access to *`model`*`_B` and other data structures. See "Interfaces for Monitoring Signals" on page 19-120 for more information.

If you have an Embedded Coder license, you can use and create custom storage classes to represent more complex data such as structures and macros. You can also use custom storage classes to export data declarations to specific generated files. For more information, see "Introduction to Custom Storage Classes" on page 23-2 and "Simulink Package Custom Storage Classes" on page 23-5.

In this section...

In this section...

Storage Classes for Signals and States

To control the code generated for signals and states, use storage class. For example, you can import or export the corresponding variable to or from the code.

You can choose from these built-in storage classes:

- `Auto`, which is the default storage class. The code generator determines signal or state storage based on optimization settings such as configuration parameters.
- `SimulinkGlobal`. The generated code contains a structured global variable for signals and another structured global variable for states that use this storage class. Each signal or state appears as a field of the appropriate structure.
- `ExportedGlobal`. Export the signal or state as a unique global variable in the generated code. The code contains an `extern` declaration for the variable.
- `ImportedExtern`. Import the signal or state as a unique global variable in the generated code. Your code must provide the variable definition.
- `ImportedExternPointer`. Import the signal or state as a unique global pointer variable in the generated code. Your code must provide the pointer variable definition.

Storage Classes for Signal Lines

To control the code generated for signals that use `Auto` storage class, on the **All Parameters** tab of the Configuration Parameters dialog box, adjust these options:

- **Signal storage reuse**
- **Reuse block outputs**
- **Enable local block outputs**

- **Eliminate superfluous local variables (expression folding)**

These configuration parameters determine, for example, whether a signal appears in the code as a local variable or whether expression folding eliminates the signal altogether.

To customize the code generation for an individual signal, specify a storage class other than `Auto`. For each of the storage classes, the table shows the variable declaration and the code generated for the Inport signal, `inSig`, of the example model `signal_examp`.

Storage Class	Declaration and Definition	Code

`Auto` (with storage optimizations on)	In `signal_examp.h`
	`typedef struct {`
	`  real_T inSig;`
	`} ExtU_signal_examp_T;`
	`extern ExtU_signal_examp_T signal_examp_U;`
		`Gain1Sig =`
		`  signal_examp_P.Gain_Gain *`
		`  signal_examp_U.inSig;`

`SimulinkGlobal`	In `signal_examp.h`
	`typedef struct {`
	`  real_T inSig;`
	`} ExtU_signal_examp_T;`
	`extern ExtU_signal_examp_T signal_examp_U;`
		`Gain1Sig =`
		`  signal_examp_P.Gain_Gain *`
		`  signal_examp_U.inSig;`

`ExportedGlobal`	In `signal_examp.c`
	`real_T inSig;`
	In `signal_examp.h`
	`extern real_T inSig;`
		`Gain1Sig = inSig *`
		`  signal_examp_P.Gain_Gain;`

`ImportedExtern`	In `signal_examp_private.h` `extern real_T inSig;`	`Gain1Sig = inSig *` `  signal_examp_P.Gain_Gain;`
`ImportedExternPointer`	In `signal_examp_private.h`	`Gain1Sig = (*inSig) *` `  signal_examp_P.Gain_Gain;`

Storage Class	Declaration and Definition	Code
	`extern real_T *inSig;`	

Storage Classes for Block States

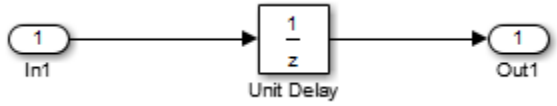
Use the `Auto` storage class for states that you do not need to interface to external code. States with `Auto` storage class are typically stored as fields of the `DWork` structure in the generated code.

You can assign a symbolic name to states that use the `Auto` storage class. If you do not supply a state name, the code generator produces one, as described in "Discrete Block State Naming in Generated Code" on page 19-160.

States with `SimulinkGlobal` storage class are stored as fields of the `DWork` structure.

Block states with storage classes other than `Auto` or `SimulinkGlobal` are stored in unstructured global variables, independent of the `Dwork` vector. Use these storage classes for states that you want to interface to external code.

For each of the storage classes, the table shows the variable declaration and initialization code generated for the Unit Delay block state in the example model `state_examp`. The block state name is specified as `udx`. The initial value is `0`.



Storage Class	Declaration and Definition	Initialization Code

`Auto`	In `state_examp.h`
	
	`typedef struct`
	`{`
	`  real_T udx;`
	`  }`
	`DW_state_examp_T;`
		`state_examp_DW.udx = 0.0;`

`SimulinkGlobal`	In `state_examp.h`
	
	`typedef struct`
	`{`
	`  real_T udx;`
	`  }`
	`DW_state_examp_T;`
		`state_examp_DW.udx = 0.0;`

Storage Class	Declaration and Definition	Initialization Code

`ExportedGlobal`	In `state_examp.c`
	`real_T udx;`
	In `state_examp.h`
	`extern real_T udx;`
		`udx = 0.0;`

`ImportedExtern`	In `state_examp_private.h` `extern real_T udx;`	`udx = 0.0;`
`ImportedExternPointer`	In `state_examp_private.h` `extern real_T *udx;`	`*udx = state_examp_U.In1;`

Use Model Data Editor to Configure Data Interface

Use the Model Data Editor to apply storage classes to Inport and Outport blocks, signal lines, and Data Store Memory blocks. Use this technique to apply the storage classes without locating the blocks and signals in the model and to configure the data interface of the model by using a single list.

To open the Model Data Editor, select **View** > **Model Data**. For information about using the Model Data Editor, see "Configure Data Properties by Using a Table" (Simulink). For an example that shows how to apply storage classes to Inport and Outport blocks, see "Design Data Interface by Configuring Inport and Outport Blocks" on page 19-134.

- For an Inport block, the Model Data Editor applies the storage class to the output signal of the block, not to the block itself.
- You can use the Model Data Editor to apply a storage class directly to a root-level Outport block or to the input signal that drives the block.

  - To store the storage class specification in the Outport block, use the **Inports/Outports** tab in the Model Data Editor. When you use this technique, the specification remains after you delete the input signal that drives the block. Use this technique to configure the model interface before you develop the internal algorithm.
  - To store the specification in the input signal that drives the block, use the **Signals** tab in the Model Data Editor or use the Signal Properties dialog box. Use this technique to use the custom storage class `Reusable` (Embedded Coder). To use

this custom storage class, you associate the signal line with a `Simulink.Signal` object in a workspace or data dictionary instead of storing the specification in the signal line.

Signal Objects for Code Generation

To specify code generation options for a signal or state in a model by using a data object:

1. Create a signal data object, which is an object of the class `Simulink.Signal` or of a subclass of `Simulink.Signal`.
2. Specify code generation options by modifying the `CoderInfo` property of the signal object.
3. Associate the signal object with a signal or state in a model diagram. For example, in a Signal Properties dialog box, or on the **State Attributes** tab of a block dialog box, specify the object name as the name of the target signal or state.
4. Generate code and build your target executable.

Signal objects have a property `CoderInfo` that contains an object of the class `Simulink.CoderInfo`. You can use the properties of the `Simulink.CoderInfo` object to specify code generation options for the target model signal or state.

For basic information about signal objects, including how to create them, see "Data Objects" (Simulink).

Create and Configure Signal Object for Code Generation

To control the code generation of a signal by creating a signal object, you can use Model Explorer.

1. In the Model Explorer **Model Hierarchy** pane, select a workspace to contain the signal object. For example, select`Base Workspace`.
2. Click Add Signal  .

   A `Simulink.Signal` object named `Sig` appears in the base workspace.
3. In the **Contents** pane, change the name of the signal object to a meaningful name. For example, name the object `mySignal` or `myState`.
4. Select the signal object in the **Contents** pane. In the **Dialog** pane, in the **Storage class** drop-down list, select a storage class such as `ExportedGlobal`. Click **Apply**.

5 Open the Signal Properties dialog box for a signal in a model, or open the **State Attributes** tab of a block dialog box. Specify **Signal name** or **State name** as `mySignal` or `myState` and click **Apply**.

6 Select the check box next to **Signal name must resolve to Simulink signal object** or **State name must resolve to Simulink signal object**. Click **OK**.

7 Generate code.

Programmatically Create and Configure Signal Object for Code Generation

At the command prompt, you can control the code generation of a signal by creating a signal object.

1 Create a `Simulink.Signal` object named `mySignal` or `myState` in the base workspace.

```
mySignal = Simulink.Signal;
```

2 Specify a storage class for the object. For example, specify the storage class `ExportedGlobal`.

```
mySignal.StorageClass = 'ExportedGlobal';
```

The `StorageClass` property is a property of the `Simulink.CoderInfo` object that resides in the `CoderInfo` property of the signal object. However, you can use the preceding syntax to access the storage class property. You can also explicitly access the property by using the syntax `mySignal.CoderInfo.StorageClass`.

3 Open the Signal Properties dialog box for a signal in a model, or open the **State Attributes** tab of a block dialog box. Specify **Signal name** or **State name** as `mySignal` or `myState` and click **Apply**.

4 Select the check box next to **Signal name must resolve to Simulink signal object** or **State name must resolve to Simulink signal object**. Click **OK**.

5 Generate code.

Apply Storage Classes Directly to Signal Lines, Block States, and Outport Blocks

Through dialog boxes, you can apply storage classes directly to signal lines and block states. You do not need a data object that you store in a workspace or data dictionary.

However, if you specify a storage class for a signal or state with this technique, you cannot use a signal object in a workspace to specify other characteristics of the signal or state, such as data type.

To apply a storage class directly to a signal line, use the Signal Properties dialog box. For a block state, use the **State Attributes** tab in the block dialog box.

1. Open the **Code Generation** tab in a Signal Properties dialog box, or the **State Attributes** tab in a block dialog box.
2. Specify a name in the **Signal name** box or the **State name** box. Click **Apply**.
3. In the **Storage class** drop-down list , select a storage class.

To apply a storage class directly to an Outport block, use the Model Data Editor. You can also use the Model Data Editor to apply custom storage classes to signals through a list that you can sort, group, and filter. See "Use Model Data Editor to Configure Data Interface" on page 19-127.

Programmatically Apply Storage Classes Directly to Signals, States, and Outport Blocks

To programmatically apply storage classes to signal lines and block states, use the function `set_param`. You can apply a storage class without creating a `Simulink.Signal` object in a workspace or data dictionary. The storage class specification is saved in the model file.

However, you can use this technique to specify only a storage class for the object. You must specify other signal or state characteristics, such as data type, in the source block dialog box. You cannot use a signal object in a workspace to specify these other characteristics.

This example shows how to programmatically apply a storage class to a signal line.

1. Open the example model `rtwdemo_secondOrderSystem`.

   ```
   rtwdemo_secondOrderSystem
   ```

2. Get a handle to the output of the block named Force: f(t).

   ```
   portHandles = get_param('rtwdemo_secondOrderSystem/Force: f(t)','PortHandles');
   outportHandle = portHandles.Outport;
   ```

3. Set the name of the corresponding signal to `ForceSignal`.

   ```
   set_param(outportHandle,'Name','ForceSignal')
   ```

4. Set the storage class of the signal to `ExportedGlobal`.

   ```
   set_param(outportHandle,'StorageClass','ExportedGlobal')
   ```

5. Generate code from the model. The code declares and defines a global variable `ForceSignal` to represent the signal.

To apply a storage class directly to an Outport block, using the function `set_param`, specify the block parameter `SignalName` to name the signal that the block represents. Use the parameter `StorageClass` to specify a storage class.

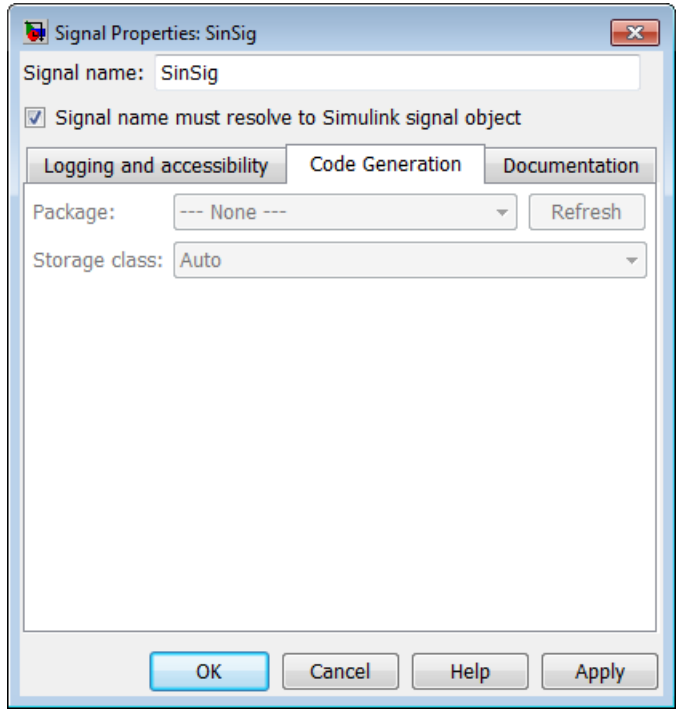
To apply a storage class to a block state, using the function `set_param`, specify the block parameter `StateIdentifier` to name the state. Use the parameter `StateStorageClass` to specify a storage class.

To apply a storage class to a data store that you define by using a Data Store Memory block, use the block parameter `StateStorageClass`. You do not need to specify a state name because the data store already has a name.

To programmatically apply a custom storage class to a signal, state, Outport block, or data store, use embedded signal objects. Custom storage classes do not affect the generated code unless you use a system target file based on `ert.tlc`, which requires Embedded Coder. You can also use this technique to set storage classes. See "Programmatically Apply Custom Storage Classes Directly to Signals, States, and Outport Blocks Using Embedded Signal Objects" on page 23-63.

Resolve Conflicts in Configuration of Signal Objects

If a signal is defined in the Signal Properties dialog box and a signal object of the same name is defined by using the command line or in the Model Explorer, the potential exists for ambiguity when the Simulink engine attempts to resolve the symbol representing the signal name. One way to resolve the ambiguity is to specify that a signal resolve to a Simulink data object. Select the **Signal name must resolve to Simulink signal object** option in the Signal Properties dialog box. You cannot specify the **Storage class** property on the **Code Generation** tab in the Signal Properties dialog box.



As the preceding figure shows, the **Storage class** menu is disabled because it is up to the `SinSig` `Simulink.Signal` object to specify its own storage class.

The signal and signal objects `SinSig` both have `SimulinkGlobal` storage class. Therefore, `SinSig` resolves to the signal object `SinSig`.

Note The rules for compatibility between block states/signal objects are identical to those given for signals/signal objects.

Related Examples

- "Access Signal, State, and Parameter Data During Execution" on page 19-3
- "Design Data Interface by Configuring Inport and Outport Blocks" on page 19-134
- "Maximize Signal Storage Optimization" (Simulink Coder)

- “Group Signals into Structures in the Generated Code Using Buses” on page 19-139
- “Control Signal Data Types” (Simulink)
- “Signal Representation in Generated Code” (Simulink Coder)
- “Storage Classes for Signals Used with Model Blocks” (Simulink Coder)
- “Control Signal and State Initialization in the Generated Code” (Simulink Coder)
- “Default Data Structures in the Generated Code” (Simulink Coder)
- “Virtualized Output Ports Optimization” on page 55-17

Design Data Interface by Configuring Inport and Outport Blocks

The data interface of a model is the means by which the model exchanges data (for example, signal values) with other, external models or systems. Customize the data interface of a model to:

- Enable integration of the generated code with your own code.
- Improve traceability and readability of the code.

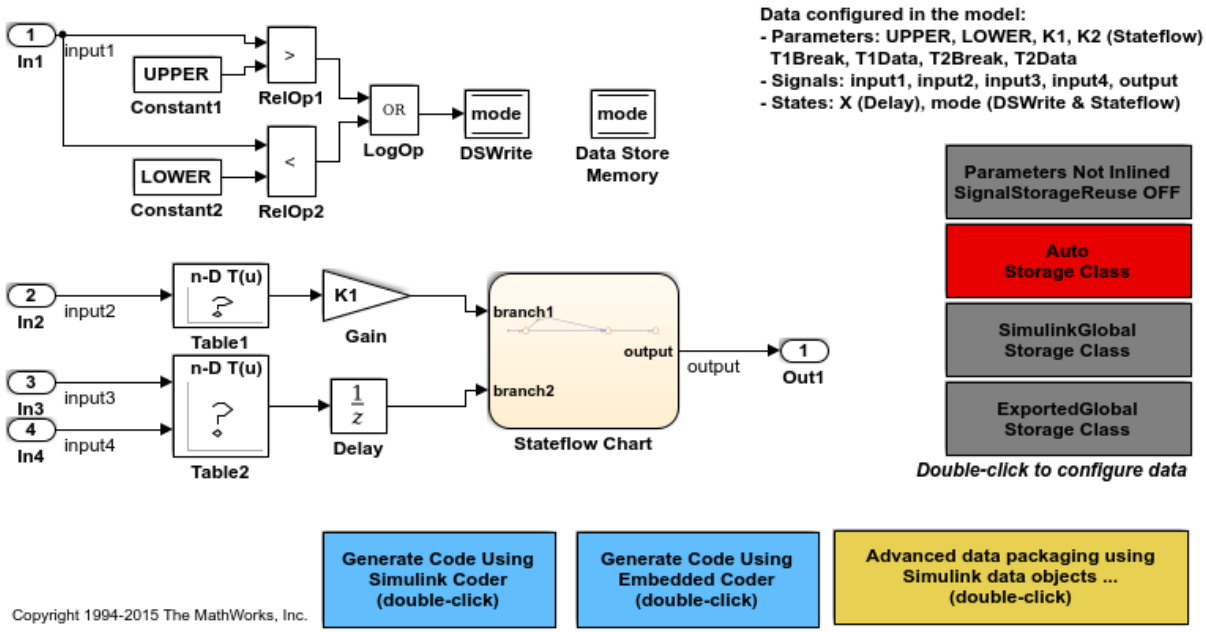
At the top level of a model, Inport and Outport blocks represent the input and output signals of the model. To customize the data interface in the generated code, configure these blocks. Early in the design process, when a model can contain unconnected Inport and Outport blocks, use this technique to specify the interface before developing the internal algorithm.

When you apply storage classes to Inport and Outport blocks, each block appears in the generated code as a field of a global structure or as a separate global variable that the generated algorithm references directly. If you have Embedded Coder, you can use function prototype control instead of storage classes to pass data into and out of the model `step` function as formal parameters. See “Control Generation of Function Prototypes”.

Design Data Interface

Open the example model rtwdemo_basicsc.

```
open_system('rtwdemo_basicsc')
```



In the model, select **View > Model Data**.

In the Model Data Editor, select the **Inports/Outports** tab. Each row in the table represents an Inport or Outport block.

Name the output signal of the Outport block labeled `Out1`. Set **Signal Name** to `output_sig`.

For each of the Inport blocks, set **Data Type** to `single` or to a different data type. Due to the data type inheritance settings that the other blocks in the model use by default, downstream signals use the same or a similar data type.

Optionally, configure other design attributes such as **Min** and **Max** (minimum and maximum values).

Set the **Change View** drop-down list to `Code`.

For the Outport block and all of the Inport blocks, set **Storage Class** to `ExportedGlobal`. To configure all of the blocks at once, select all of the rows in the table.

To configure the blocks and signals, you can use these commands at the command prompt.

```
temp = Simulink.Signal;
temp.StorageClass = 'ExportedGlobal';

portHandles = get_param('rtwdemo_basicsc/In1','portHandles');
outPortHandle = portHandles.Outport;
set_param(outPortHandle,'SignalObject',temp);

portHandles = get_param('rtwdemo_basicsc/In2','portHandles');
outPortHandle = portHandles.Outport;
set_param(outPortHandle,'SignalObject',temp);

portHandles = get_param('rtwdemo_basicsc/In3','portHandles');
outPortHandle = portHandles.Outport;
set_param(outPortHandle,'SignalObject',temp);

portHandles = get_param('rtwdemo_basicsc/In4','portHandles');
outPortHandle = portHandles.Outport;
set_param(outPortHandle,'SignalObject',temp);

set_param('rtwdemo_basicsc/Out1','SignalName','output_sig',...
    'SignalObject',temp)
```

Generate code from the model.

```
rtwbuild('rtwdemo_basicsc');

### Starting build procedure for model: rtwdemo_basicsc
### Successful completion of build procedure for model: rtwdemo_basicsc
```

View the generated file `rtwdemo_basicsc.c`. Because you applied the storage class `ExportedGlobal` to the Inport and Outport blocks, the code creates separate global variables that represent the inputs and the output.

```
file = fullfile('rtwdemo_basicsc_grt_rtw','rtwdemo_basicsc.c');
rtwdemodbtype(file,'/* Exported block signals */','real32_T output_sig;',1,1)


/* Exported block signals */
real32_T input1;                       /* '<Root>/In1' */
real32_T input2;                       /* '<Root>/In2' */
real32_T input3;                       /* '<Root>/In3' */
real32_T input4;                       /* '<Root>/In4' */
```

```
real32_T output_sig;                   /* '<Root>/Out1' */
```

The generated algorithm in the model `step` function directly references these global variables to calculate and store the output signal value, `output_sig`.

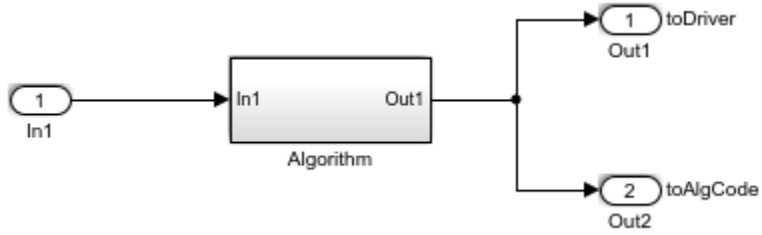
While you use the Model Data Editor to configure the interface of a system, consider using the interface display to view the system inputs and outputs (Inport and Outport blocks) at a high level. See "Configure Data Interface for Component" (Simulink).

Route Signal Data to Multiple Outputs

You can route a single signal to multiple Outport blocks and apply a different storage class to each Outport. For example, use this technique to send signal data to a custom function (such as a device driver) and to a global variable that your custom algorithmic code can use:

1. Branch the target signal line to each Outport block.
2. For more efficient code, set the storage class of the target signal line to `Auto` (the default). Optimizations can then eliminate the signal line from the generated code.
3. Use the Model Data Editor to apply the custom storage class `GetSet` to one Outport block and `ExportToFile` to the other Outport block. Apply a signal name to each block.

```
open_system('ex_route_sig')
```



Limitations

You cannot apply a storage class to an Outport block if the input to the block has a variable size. Instead, apply the storage class to the signal line.

Related Examples

- "Analyze the Generated Code Interface" on page 35-21

- “Trace Connections Using Interface Display” (Simulink)
- “Interface Design” (Simulink)
- “Signal Representation in Generated Code” on page 19-112
- “Configure Generated Code According to Interface Control Document” on page 23-112
- “Control Generation of Function Prototypes” on page 26-2
- “Configure Data Properties by Using a Table” (Simulink)
- “Conform to Coding Standards by Replacing and Renaming Data Types” on page 21-22

Group Signals into Structures in the Generated Code Using Buses

Buses in a model represent multiple signals as a single signal line. You can use nonvirtual buses to create signal structures in the generated code.

You can generate flat and nested structures. See “Structures of Signals” on page 13-87.

For basic information about buses in models, see “Buses” (Simulink).

Import or Export Structure Variable and Definition

This example shows how to export the definition of a bus type and the declaration of a nonvirtual bus signal in generated code. You can control the bus type definition and the bus signal declaration in the generated code independently of each other.

Explore Example Model

1. Open the model rtwdemo_slbus.

   The model creates a `Simulink.Bus` object `BusObject` in the base workspace.
2. Update the model diagram to display thick lines for composite signals, including buses.

   The bus signal `S1` uses the bus type that `BusObject` defines.

Control Scope of Bus Type

To control code generated for a bus type that a `Simulink.Bus` object defines, adjust the code generation settings for the object.

The generated code represents nonvirtual bus types with `struct` definitions. The `struct` definitions contain fields corresponding to the elements of the bus.

1. At the command prompt, specify the `DataScope` property of `BusObject` as `'Exported'`.

   ```
   BusObject.DataScope = 'Exported';
   ```
2. Specify the `HeaderFile` property of the object as `'myBusTypeHdr.h'`.

   ```
   BusObject.HeaderFile = 'myBusTypeHdr.h';
   ```

Control Scope of Bus Signal

You can use a `Simulink.Signal` object to control code generated for signals in a model, including composite signals such as buses. To control code generated for a nonvirtual bus signal, apply a storage class to the bus.

1. View the model in the Simulink Editor.
2. In the signal properties dialog box for the bus signal `S1`, select the option **Signal name must resolve to Simulink.Signal object**.
3. At the command prompt, create a `Simulink.Signal` object to represent the bus signal `S1`. Specify the data type of the object as `BusObject`.

   ```
   S1 = Simulink.Signal;
   S1.DataType = 'Bus: BusObject';
   ```

4. Specify the storage class of the object as `ExportedGlobal`.

   ```
   S1.StorageClass = 'ExportedGlobal';
   ```

For more information about applying storage classes to signals, see "Control Signals and States in Code by Applying Storage Classes" on page 19-123.

Generate and Inspect Code

1. In the Simulink Editor, double-click the blue box labeled **Generate Code Using Simulink Coder**.
2. In the code generation report, view the generated file `myBusTypeHdr.h`. The code in the header file represents the bus type `BusObject` with a `struct` definition.

   ```
   typedef struct {
     real_T temperature;
     real_T heat;
     real_T pressure[20];
   } BusObject;
   ```

3. View the file `rtwdemo_slbus.c`. The code exports the declaration of the signal `S1` and uses the structure type `BusObject` in the declaration.

   ```
   /* Exported block signals */
   BusObject S1;
   ```

Generate Code That Reuses `struct` Types from Existing C Code

If your existing C code exchanges data through structures, you can generate code that packages and accesses signal data according to the existing `struct` type definitions. This technique enables you to match interfaces between the bodies of code and to compile the code into a single application. For an example, see "Exchange Structured and Enumerated Data Between Generated and External Code" on page 21-28.

Arrays of Structures

You can further package multiple consistent bus signals into an array of buses. The array of buses appears in the generated code as an array of structures. To create arrays of buses, see "Combine Buses into an Array of Buses" (Simulink).

Structure Padding

By default, the code generator does not explicitly add padding fields to structure types. Structure types can appear in the generated code through, for example, the default data structures (see "Default Data Structures in the Generated Code" (Simulink Coder)), `Simulink.Bus` objects, and parameter structures that you use in a model.

However, when you use a code replacement library with Embedded Coder, you can specify data alignment (including structure padding) as part of the replacement library. For more information, see "Provide Data Alignment Specifications for Compilers" on page 51-135.

See Also

Simulink.Bus | `Simulink.Signal`

Related Examples

- "Generate Efficient Code for Bus Signals" on page 19-142
- "Specify Sample Times for Signal Elements" (Simulink)
- "Signal Representation in Generated Code" on page 19-112
- "Control Signals and States in Code by Applying Storage Classes" on page 19-123
- "Buses" (Simulink)
- "Exchange Structured and Enumerated Data Between Generated and External Code" on page 21-28

Generate Efficient Code for Bus Signals

In this section...

In a model, you use bus signals to package multiple signals together into a single signal line. You can create virtual or nonvirtual bus signals. The representation in the generated code depends on:

- For a virtual bus, the generated code appears as if the bus did not exist.
- Generated code for a nonvirtual bus represents the bus data with a structure. When you want to trace the correspondence between the model and the code, the use of a structure in the generated code can be helpful. To generate structures using nonvirtual bus signals, see "Group Signals into Structures in the Generated Code Using Buses" on page 19-139.

For general information about buses, see and "Virtual and Nonvirtual Buses" (Simulink).

To generate efficient code from models that contain bus signals, eliminate unnecessary data copies by following best practices as you construct the model.

Code Efficiency for Bus Signals

When you use buses in a model for which you intend to generate code:

- Setting bus diagnostic configuration parameters can make model development easier.
- The bus implementation techniques, and the choice of a nonvirtual or virtual bus, can influence the speed, size, and clarity of the generated code.
- Some useful bus implementation techniques are not immediately obvious.

When you work with buses, these guidelines help you to improve the results. The guidelines describe techniques to:

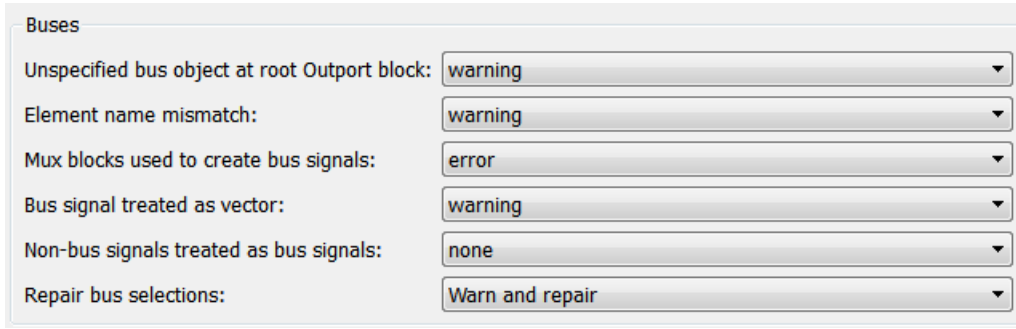
- Simplify the layout of the model.
- Increase the efficiency of generated code.
- Define data structures for function (subsystem) interfaces.

- Define data structures that match existing data structures in external C code.

There are some trade-offs among speed, size, and clarity. For example, the code for nonvirtual buses is easier to read because the buses appear in the code as structures, but the code for virtual buses is faster because virtual buses do not require copying signal data. Apply some of the guidelines based on where you are in the application development process.

Set Bus Diagnostics

Simulink provides diagnostics that you can use to optimize bus usage. Set the following values on the **Configuration Parameters** > **Diagnostics** > **Connectivity** pane.



Optimize Virtual and Nonvirtual Buses

Virtual buses are graphical conveniences that do not affect generated code. As a result, the code generation engine is able to fully optimize the signals in the bus. Use virtual buses rather than nonvirtual buses wherever possible. You can convert between virtual and nonvirtual buses by using Signal Conversion blocks. In some cases, Simulink automatically converts a virtual bus to a nonvirtual bus when required. For example, a Stateflow chart converts an input virtual bus to a nonvirtual bus.

To bundle function-call signals, you must use a virtual bus.

You must use nonvirtual buses for:

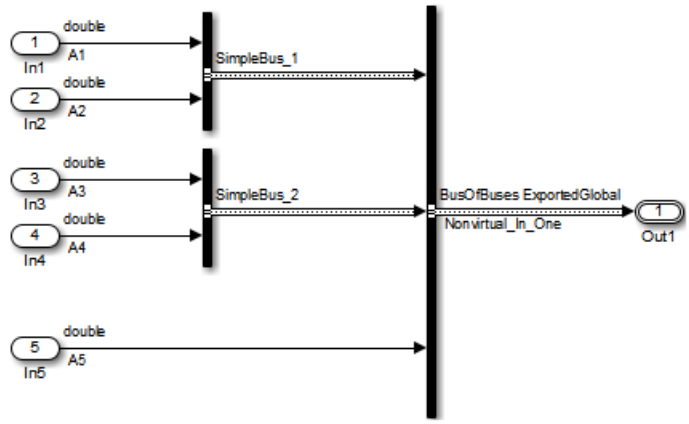
- Nonauto storage classes
- Generating a specific structure from the bus
- Root-level Inport or Outport blocks when the bus has mixed data types

Avoid Nonlocal Nested Buses in Nonvirtual Buses

Buses can contain subordinate buses. To generate efficient code, set the storage classes of subordinate buses to `Auto`. Setting the storage class to `Auto` eliminates:

- Allocation of redundant memory for the subordinate bus signal and for the parent bus signal
- Additional copy operations (copying data to the subordinate bus, and then copying from the subordinate bus to the final bus)

This model contains nonvirtual bus signals. The subordinate bus signals `Sub_Bus_1` and `Sub_Bus_2` use the storage class `Auto`.



The generated code algorithm efficiently assigns the input signal data to the bus signals.

```
void ex_nonvirtual_buses_step(void)
{
  Nonvirtual_In_One.SimpleBus_1.A1 = A1;
  Nonvirtual_In_One.SimpleBus_1.A2 = A2;
  Nonvirtual_In_One.SimpleBus_2.A3 = A3;
  Nonvirtual_In_One.SimpleBus_2.A4 = A4;
  Nonvirtual_In_One.A5 = A5;
}
```

See Also

Simulink.Bus

Related Examples

- "Group Signals into Structures in the Generated Code Using Buses" on page 19-139
- "Specify Sample Times for Signal Elements" (Simulink)

Maximize Signal Storage Optimization

The value of the "Maximum stack size (bytes)" (Simulink) parameter, on the **Optimization > Signals and Parameters** pane of the Configuration Parameters dialog box constrains the use of stack space used by local block output variables. The command-line equivalent for this parameter is `MaxStackSize`. If the accumulated size of variables in local memory exceeds `MaxStackSize`, the product places subsequent local variables in global memory space.

If it is important that you maximize potential for signal storage optimization, then set `MaxStackSize` to accommodate the size and number of signals in your model. This minimizes overflow into global memory space and maximizes use of local memory. Local variables offer more optimization potential through mechanisms such as expression folding and buffer reuse. See "Customize Stack Space Allocation" (Simulink Coder) for more information.

Related Examples

- "Customize Stack Space Allocation" (Simulink Coder)
- "Signal Representation in Generated Code" (Simulink Coder)

Control Signal and State Initialization in the Generated Code

To initialize signals and discrete states with custom values for simulation and code generation, you can use signal objects and block parameters. Data initialization increases application reliability and is a requirement of safety critical applications. Initializing signals for both simulation and code generation can expedite transitions between phases of Model-Based Design.

For basic information about specifying initial values for signals and discrete states in a model, see “Initialize Signals and Discrete States” (Simulink).

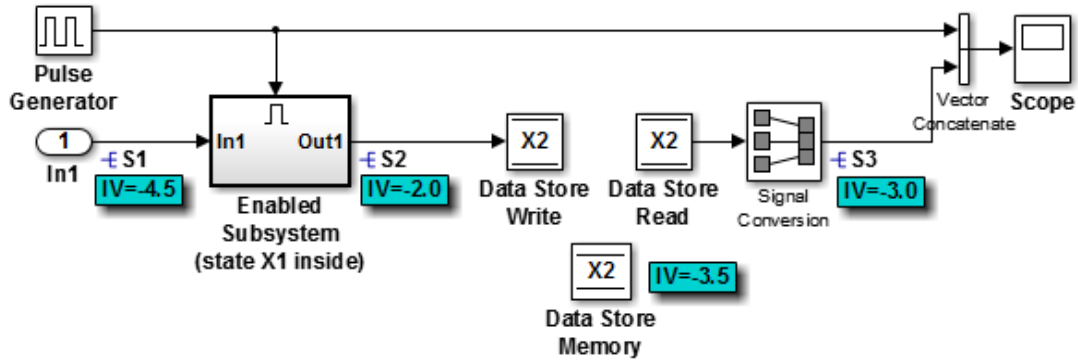
Signal and State Initialization in the Generated Code

The initialization behavior for code generation is the same as that for model simulation with the following exceptions:

- RSim executables can use the **Data Import/Export** pane of the Configuration Parameters dialog box to load input values from MAT-files. GRT and ERT executables cannot load input values from MAT-files.
- The initial value for a block output signal or root level input or output signal can be overwritten by an external (calling) program.
- Setting the initial value for persistent signals is relevant if the value is used or viewed by an external application.

When you generate code, initialization statements are placed in *`model`*`.c` or *`model`*`.cpp` in the model's initialize code.

For example, consider the model rtwdemo_sigobj_iv.



If you create and initialize signal objects in the base workspace, the code generator places initialization code for the signals in the file `rtwdemo_sigobj_iv.c` under the `rtwdemo_sigobj_iv_initialize` function, as shown below.

```
 /* Model initialize function */

     void rtwdemo_sigobj_iv_initialize(void)

     {
        .
        .
        .
 /* exported global signals */
 S3 = -3.0;

 S2 = -2.0;
        .
        .
        .
/* exported global states */
X1 = 0.0;
X2 = 0.0;

/* external inputs */
S1 = -4.5;
        .
        .
        .
```

The following code shows the initialization code for the enabled subsystem's Unit Delay block state `X1` and output signal `S2`.

```
void MdlStart(void) {
          .
          .
          .
 /* InitializeConditions for UnitDelay: '<S2>/Unit Delay' */
  X1 = aa1;

 /* Start for enable system: '<Root>/Enabled Subsystem (state X1 inside)' */

  /* virtual outports code */

  /* (Virtual) Outport Block: '<S2>/Out1' */

  S2 = aa2;

}
```

For an enabled subsystem, the initial value is also used as a reset value if the subsystem's Outport block parameter **Output when disabled** is set to `reset`. The following code from `rtwdemo_sigobj_iv.c` shows the assignment statement for `S3` as it appears in the model output function `rtwdemo_sigobj_iv_output`.

```
/* Model output function */

static void rtwdemo_sigobj_iv_output(void)
```

```
{
        .
        .
        .
 /* Disable for enable system: '<Root>/Enabled Subsystem (state X1 inside)' */

    /* (Virtual) Outport Block: '<S2>/Out1' */

    S2 = aa2;
```

Generate Tunable Initial Conditions

You can represent initial conditions for signals and states by creating tunable global variables in the generated code. These variables allow you to restart an application by using initial conditions that are stored in memory.

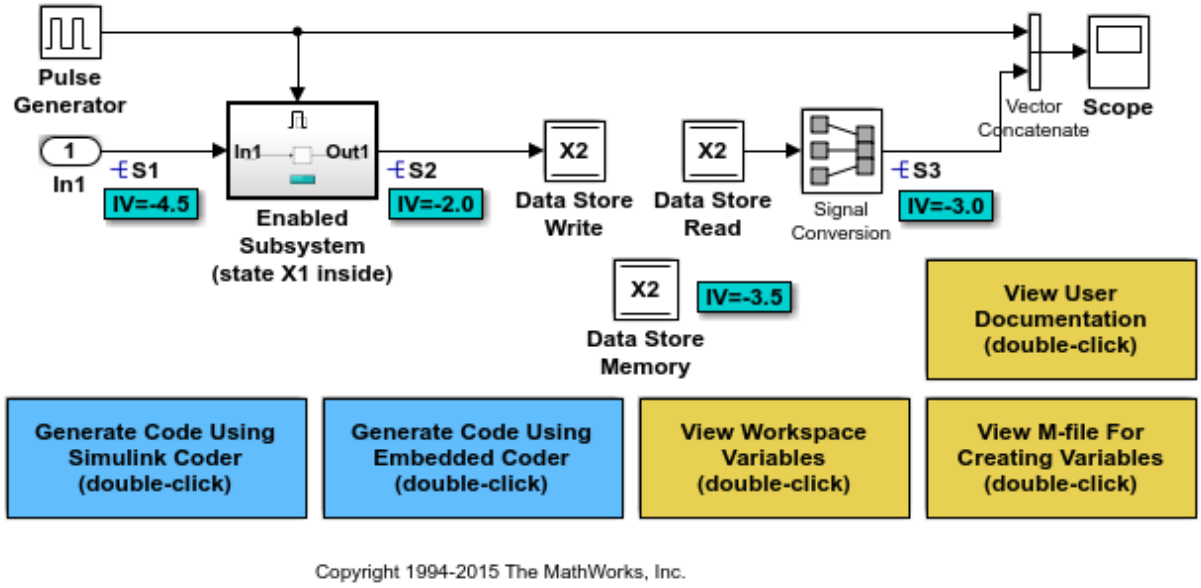
If you set **Configuration Parameters > Optimization > Signals and Parameters > Default parameter behavior** to `Tunable`, initial conditions appear as tunable fields of the global parameters structure.

Whether you set **Default parameter behavior** to `Tunable` or `Inlined`, you can use a tunable parameter to specify the `InitialValue` property of a signal object or the **Initial condition** parameter of a block. For basic information about tunable parameters, see "Block Parameter Representation in the Generated Code" (Simulink Coder).

This example shows how to use tunable parameters to specify initial conditions for signals and states.

Explore Example Model

Open the example model rtwdemo_sigobj_iv. The signal `S2` uses a `Simulink.Signal` object in the base workspace.



Double-click the `Simulink.Signal` object `S2` to view its properties. The **Initial value** property is set to `aa2`. The object uses the variable `aa2` to specify an initial condition for the signal `S2`. The **Storage class** property is set to `ExportedGlobal`. To use a `Simulink.Signal` object to initialize a signal, the signal object must use a storage class other than `Auto` or `SimulinkGlobal`.

On the **Optimization > Signals and Parameters** pane in the Configuration Parameters dialog box, click **Configure**. The variable `aa2` is a tunable parameter that uses the storage class `ExportedGlobal`.

In the model, open the Enabled Subsystem. In the Outport block dialog box, the parameter **Output when disabled** is set to `reset`. When the subsystem becomes disabled, the output signal `S2` resets to the initial value `aa2`.

Open the Unit Delay block dialog box. On the **State Attributes** tab, the **State name** box is set to `X1`.

Open the Enable block dialog box. The parameter **States when enabling** is set to `reset`. When the subsystem transitions from a disabled state to an enabled state, it resets internal block states, such as `X1`, to their initial values.

In the base workspace, double-click the `Simulink.Signal` object `X1` to view its properties. The **Initial value** property is set to `aa1`.

Double-click the `Simulink.Parameter` object `aa1` to view its properties. The **Storage class** property is set to `ExportedGlobal`. You can generate tunable initial conditions for block states by using tunable parameters such as `aa1` and `Simulink.Signal` objects such as `X1`.

Generate and Inspect Code

Generate code with the example model.

```
### Starting build procedure for model: rtwdemo_sigobj_iv
### Successful completion of build procedure for model: rtwdemo_sigobj_iv
```

In the code generation report, view the file `rtwdemo_sigobj_iv.c`. The code uses global variables to represent the block state `X1` and the signal `S2`.

```
/* Exported block states */
real_T X1;                             /* '<S2>/Unit Delay' */

/* Exported block signals */
real_T S1;                             /* '<Root>/In1' */
real_T S3;                             /* '<Root>/Signal Conversion' */
real_T S2;                             /* '<S2>/Unit Delay' */
```

The code uses global variables to represent the tunable parameters `aa1` and `aa2`.

```
/* Exported block parameters */
real_T aa1 = -2.5;                     /* Variable: aa1

real_T aa2 = -2.0;                     /* Variable: aa2
```

The model initialization function uses the tunable parameter `aa1` to initialize the state `X1`. The function also uses the tunable parameter `aa2` to initialize the signal `S2`.

```
  /* SystemInitialize for Enabled SubSystem: '<Root>/Enabled Subsystem (state X1 inside
  /* InitializeConditions for UnitDelay: '<S2>/Unit Delay' */
  X1 = aa1;

  /* SystemInitialize for Outport: '<S2>/Out1' */
  S2 = aa2;
```

In the model step function, when the Enabled Subsystem transitions from a disabled state to an enabled state, the Unit Delay block state `X1` resets to its initial value.

```
if (rtb_PulseGenerator > 0) {
  if (!rtwdemo_sigobj_iv_DW.EnabledSubsystemstateX1inside_M) {
    /* InitializeConditions for UnitDelay: '<S2>/Unit Delay' */
    X1 = aa1;
    rtwdemo_sigobj_iv_DW.EnabledSubsystemstateX1inside_M = true;
  }
```

If the Enabled Subsystem becomes disabled during code execution, the algorithm uses the tunable initial condition `aa2` to set the value of the signal `S2`.

```
} else {

  if (rtwdemo_sigobj_iv_DW.EnabledSubsystemstateX1inside_M) {
    /* Disable for Outport: '<S2>/Out1' */
    S2 = aa2;
    rtwdemo_sigobj_iv_DW.EnabledSubsystemstateX1inside_M = false;
  }
}
```

Generate Tunable Initial Condition Structure for Bus Signal

When you use a MATLAB® structure to specify initialization values for the signal elements in a bus, you can create a tunable global structure in the generated code.

If you set **Configuration Parameters > Optimization > Signals and Parameters > Default parameter behavior** to `Tunable`, the initial condition appears as a tunable substructure of the global parameters structure.

Whether you set **Default parameter behavior** to `Tunable` or `Inlined`, you can specify the initial condition by using a tunable `Simulink.Parameter` object whose value is a structure. If you apply a storage class other than `Auto` to the parameter object, the structure is tunable in the generated code.

To generate efficient code by avoiding data type mismatches between the structure and the bus signal, use either:

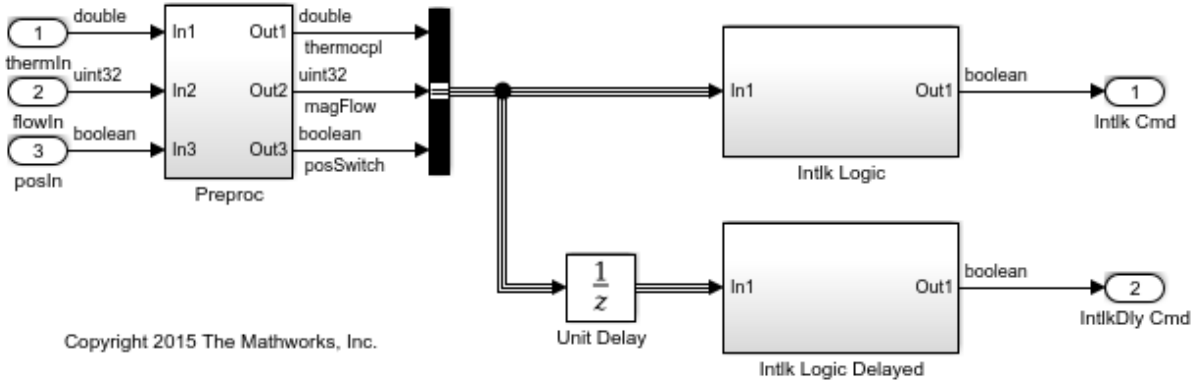
- Typed expressions to specify the values of the structure fields. Match the data type of each field with the data type of the corresponding signal element.
- A `Simulink.Bus` object to control the data types of the structure fields and the signal elements.

For basic information about using structures to initialize bus signals, and to decide how to control field data types, see "Specify Initial Conditions for Bus Signals" (Simulink).

Generate Tunable Initial Condition Structure

This example shows how to use a tunable structure parameter to initialize a virtual bus signal.

Open the example model rtwdemo_tunable_init_struct.



In the Inport block dialog boxes, open the **Signal Attributes** tab. Each Inport uses a different output data type.

Open the Bus Creator block dialog box. The block output is a virtual bus.

In the Configuration Parameters dialog box, open the **Optimization > Signals and Parameters** pane. The configuration parameter **Default parameter behavior** is set to `Tunable`. By default, block parameters, including initial conditions, appear in the generated code as tunable fields of the global parameters structure.

Open the Unit Delay block dialog box. Set **Initial condition** to a structure that specifies an initial condition for each of the three signal elements. To generate efficient code,

match the data types of the structure fields with the data types of the corresponding signal elements.

```
set_param('rtwdemo_tunable_init_struct/Unit Delay','InitialCondition',...
'struct(''thermocpl'',15.23,''magFlow'',uint32(79),''posSwitch'',false)')
```

Generate code from the example model.

```
### Starting build procedure for model: rtwdemo_tunable_init_struct
### Successful completion of build procedure for model: rtwdemo_tunable_init_struct
```

In the code generation report, view the file `rtwdemo_tunable_init_struct_types.h`. The code defines a structure type whose fields use the data types that you specified in the `struct` expression.

```
#ifndef DEFINED_TYPEDEF_FOR_struct_mqGi1jsItEOG7cf1bNqMu_
#define DEFINED_TYPEDEF_FOR_struct_mqGi1jsItEOG7cf1bNqMu_

typedef struct {
  real_T thermocpl;
  uint32_T magFlow;
  boolean_T posSwitch;
} struct_mqGi1jsItEOG7cf1bNqMu;
```

View the file `rtwdemo_tunable_init_struct.h`. The `struct` type definition of the global parameters structure contains a substructure, `UnitDelay_InitialCondition`, which represents the **Initial condition** parameter of the Unit Delay block.

```
struct P_rtwdemo_tunable_init_struct_T_ {
  struct_mqGi1jsItEOG7cf1bNqMu UnitDelay_InitialCondition;/* Mask Parameter: UnitDelay_
```

View the file `rtwdemo_tunable_init_struct_data.c`. This source file allocates memory for the global parameters structure. The substructure `UnitDelay_InitialCondition` appears.

```
/* Block parameters (auto storage) */
P_rtwdemo_tunable_init_struct_T rtwdemo_tunable_init_struct_P = {
  {
    15.23,
    79U,
    0
  },                                   /* Mask Parameter: UnitDelay_InitialCondition
```

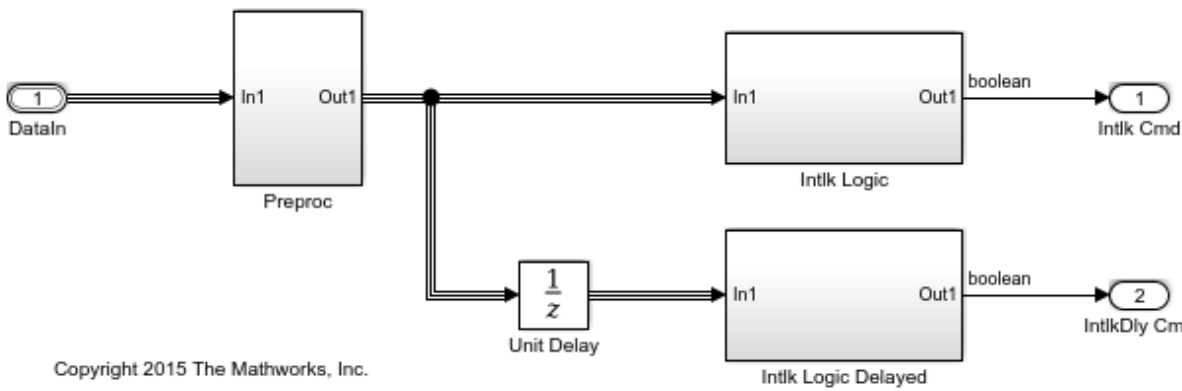
View the file `rtwdemo_tunable_init_struct.c`. The model initialization function uses the fields of the substructure to initialize the block states.

```
/* InitializeConditions for UnitDelay: '<Root>/Unit Delay' */
rtwdemo_tunable_init_struct_DW.UnitDelay_1_DSTATE =
  rtwdemo_tunable_init_struct_P.UnitDelay_InitialCondition.thermocpl;
rtwdemo_tunable_init_struct_DW.UnitDelay_2_DSTATE =
  rtwdemo_tunable_init_struct_P.UnitDelay_InitialCondition.magFlow;
rtwdemo_tunable_init_struct_DW.UnitDelay_3_DSTATE =
  rtwdemo_tunable_init_struct_P.UnitDelay_InitialCondition.posSwitch;
```

Use Bus Object to Specify Data Types

If you create a bus object, you can use it to specify the data type of the bus signal and the tunable initial condition structure. Before code generation, the `Simulink.Parameter` object casts the values of the structure fields to the data types of the signal elements. For basic information about creating bus objects and using them in models, see "When to Use Bus Objects" (Simulink).

Open the example model rtwdemo_init_struct_busobj.



In the base workspace, double-click the `Simulink.Bus` object `ComponentData`. The object defines three signal elements: `thermocpl`, `magFlow`, and `posSwitch`. The elements each use a different data type.

Open the block dialog box for the Inport block DataIn. The output data type is set to `Bus: ComponentData`.

Create the tunable structure parameter `initStruct`. You can specify the field values by using untyped expressions. To improve readability, specify the field `posSwitch` with a Boolean value. Specify the `DataType` property of the parameter object as `'Bus: ComponentData'`.

```
initStruct = struct(...
    'thermocpl',15.23,...
    'magFlow',79,...
    'posSwitch',false ...
    );

initStruct = Simulink.Parameter(initStruct);
initStruct.StorageClass = 'ExportedGlobal';
initStruct.DataType = 'Bus: ComponentData';
```

In the Unit Delay block dialog box, specify **Initial condition** as `initStruct`.

Generate code from the example model.

```
### Starting build procedure for model: rtwdemo_init_struct_busobj
### Successful completion of build procedure for model: rtwdemo_init_struct_busobj
```

In the code generation report, view the file `rtwdemo_init_struct_busobj_types.h`. The code creates a structure type `ComponentData` whose fields use the data types in the bus object.

```
#ifndef DEFINED_TYPEDEF_FOR_ComponentData_
#define DEFINED_TYPEDEF_FOR_ComponentData_

typedef struct {
  real_T thermocpl;
  uint32_T magFlow;
  boolean_T posSwitch;
} ComponentData;
```

View the file `rtwdemo_init_struct_busobj.c`. The code creates a global variable to represent the tunable parameter object `initStruct`.

```
/* Exported block parameters */
ComponentData initStruct = {
  15.23,
  79U,
```

```
  0
} ;                                    /* Variable: initStruct
```

The model initialization function uses the structure fields to initialize the block states.

```
  /* InitializeConditions for UnitDelay: '<Root>/Unit Delay' */
  rtwdemo_init_struct_busobj_DW.UnitDelay_1_DSTATE = initStruct.thermocpl;
  rtwdemo_init_struct_busobj_DW.UnitDelay_2_DSTATE = initStruct.magFlow;
  rtwdemo_init_struct_busobj_DW.UnitDelay_3_DSTATE = initStruct.posSwitch;
```

To change the data type of any of the three signal elements, specify the new type in the bus object. The signal element in the model uses the new type. Before simulation and code generation, the parameter object `initStruct` casts the corresponding structure field to the new type.

See Also

State Reader | State Writer

Related Examples

- “Initialization Behavior Summary for Signal Objects” (Simulink)
- “Specify Initial Conditions for Bus Signals” (Simulink)
- “Control Signals and States in Code by Applying Storage Classes” on page 19-123
- “Generate Code That Responds to Initialize, Reset, and Terminate Events” on page 9-2
- “Block Parameter Representation in the Generated Code” (Simulink Coder)
- “Signal Representation in Generated Code” on page 19-112
- “Initialization of Signal, State, and Parameter Data in the Generated Code” on page 19-165
- “Remove Initialization Code” on page 56-3

Continuous Block State Naming in Generated Code

In this section...

To determine the variable or field name generated for a continuous block state, do one of the following:

- Use a default name created by the code generator.
- In the block parameter dialog box, in the **State name** field, specify a name.

These names apply to the following continuous blocks:

- Integrator
- Integrator Limited
- Second-Order Integrator
- Second-Order Integrator Limited
- PID Controller
- PID Controller (2DOF)
- State Space
- Transfer Fcn
- Variable Time Delay
- Variable Transport Delay
- Zero-Pole

To name states for other types of blocks, see "Discrete Block State Naming in Generated Code" on page 19-160.

Default Block State Naming Convention

If you do not define a symbolic name for a block state, the code generator uses the following default naming convention:

```
Name#_CSTATE
```

- `Name` is the name of the block type, such as `Integrator`. If you edit the block type name displayed in the model, the code generator uses that name in the identifier.
- `#` is a unique ID number (#) assigned by the code generator if multiple instances of the same block type appear in the model. The ID number is appended to `Name`.
- `_CSTATE` is a suffix that is appended to the name and ID number.

For example, if you do not specify a state name for an Integrator block, the code generator creates the identifier:

```
rtX.Integrator_CSTATE = rtP.Integrator_IC;
```

If you specify, in the **State Name** field, `'myintegratorblockname'` for an Integrator block, the code generator creates this identifier:

```
rtX.myintegratorblockname = rtP.Integrator_IC;
```

Define User Block State Names

In the block parameters dialog box, you can define your own symbolic name for a block state.

1. In your model, double-click the block.
2. At the bottom of the block parameters dialog box, in the **State Name** field, enter the name that you want to use for the state.
3. Click **OK** or **Apply** to accept the value.
4. Generate code and examine it to see the new identifier name.

Related Examples

- "Control Signals and States in Code by Applying Storage Classes" (Simulink Coder)
- "Discrete Block State Naming in Generated Code" on page 19-160

Discrete Block State Naming in Generated Code

In this section...

To determine the variable or field name generated for a block's state, you can:

- Use a default name generated by the code generator
- Define a symbolic name by using the **State name** field of the **State Attributes** tab in a block dialog box

To name states for certain continuous blocks, see “Continuous Block State Naming in Generated Code” on page 19-158.

For certain discrete block types, you can control how block states in your model are stored and represented in the generated code. In a block dialog box, using the **State Attributes** tab, you can:

- Assign symbolic names to block states in generated code.
- Control whether states declared in generated code are interfaceable (visible) to externally written code.
- Specify that states be stored in locations declared by externally written code.

Discrete blocks that you can control how block states are stored and represented in the generated code are:

- Discrete Filter
- Discrete PID Controller
- Discrete PID Controller (2DOF)
- Discrete State-Space
- Discrete-Time Integrator
- Discrete Transfer Fcn
- Discrete Zero-Pole
- Memory
- Unit Delay

- Delay

These blocks require persistent memory to store values representing the state of the block between consecutive time intervals. By default, such values are stored in a *data type work vector*. This vector is usually referred to as the DWork vector. It is represented in generated code as `model_DWork`, a global data structure.

If you want to interface a block state to your handwritten code, you can specify that the state is stored to a location other than the `DWork` vector. You do this by assigning a storage class to the block state.

Default Block State Naming Convention

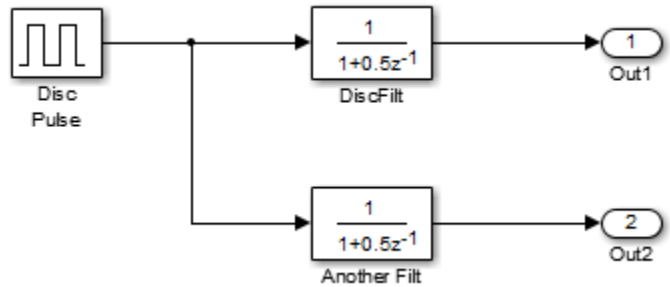
If you do not define a symbolic name for a block state, the code generator uses the following default naming convention:

```
BlockType#_DSTATE
```

where

- `BlockType` is the name of the block type (for example, Discrete_Filter). If you edit the block type name displayed in the model, the code generator uses that name in the identifier.
- `#` is a unique ID number (#) assigned by the code generator if multiple instances of the same block type appear in the model. The ID number is appended to `BlockType`.
- `_DSTATE` is a suffix that is appended to the block type and ID number.

For example, consider the model shown in the next figure.



Model with Two Discrete Filter Block States

Examine the code generated for the states of the two Discrete Filter blocks. Assume that:

- Neither blocks state has a user-defined name.
- The upper Discrete Filter block has `Auto` storage class (and is therefore stored in the `DWork` vector).
- The lower Discrete Filter block has `ExportedGlobal` storage class.

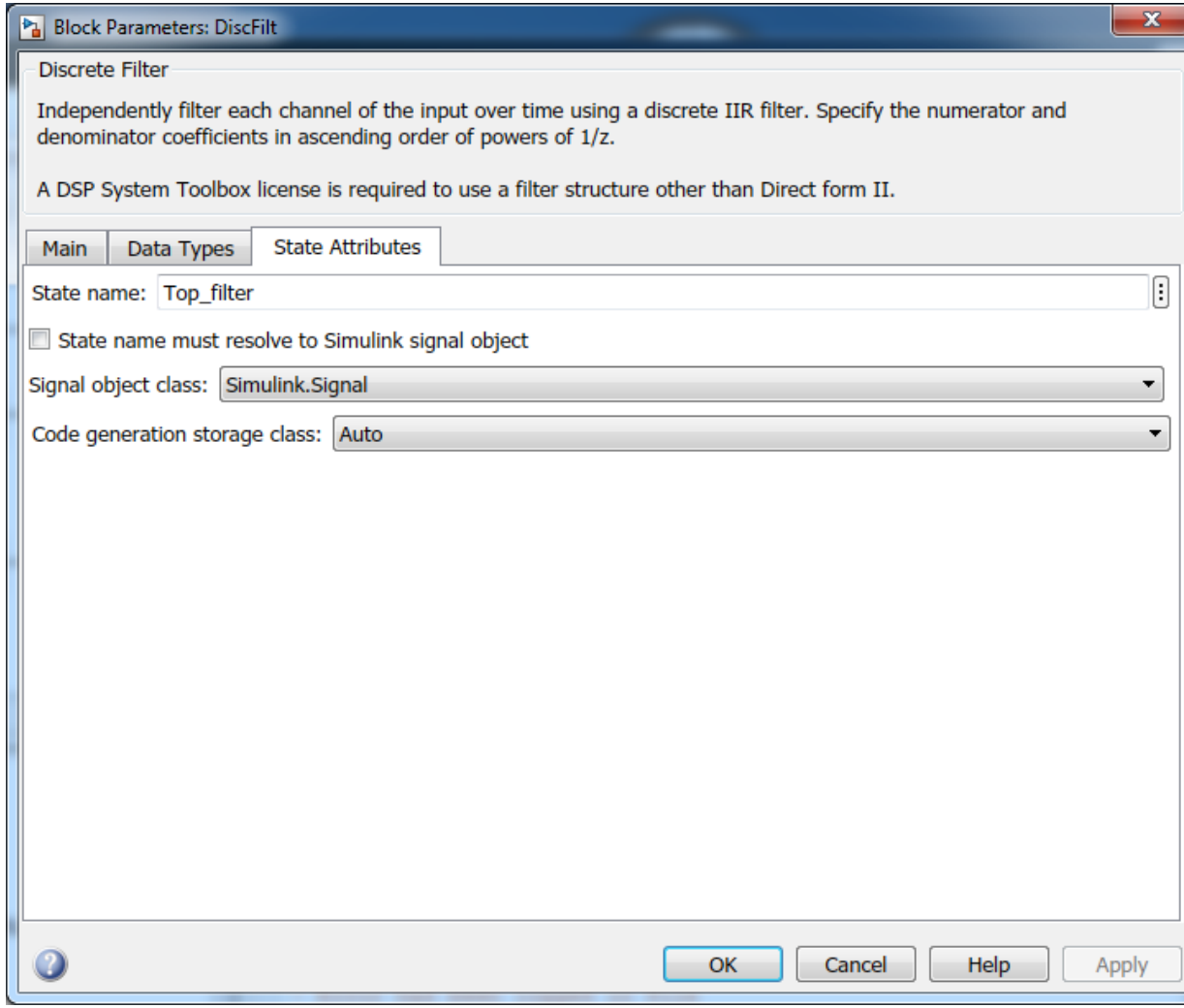
The states of the two Discrete Filter blocks are stored in `DWork` vectors, initialized as shown in the code:

```
/* data type work */
disc_filt_states_M->Work.dwork = ((void *)
&disc_filt_states_DWork);
    (void)memset((char_T *) &disc_filt_states_DWork, 0,
     sizeof(D_Work_disc_filt_states));
    {
      int_T i;
      real_T *dwork_ptr = (real_T *)
&disc_filt_states_DWork.DiscFilt_DSTATE;

      for (i = 0; i < 2; i++) {
        dwork_ptr[i] = 0.0;
      }
    }
```

Define User Block State Names

Using the **State Attributes** tab of a block dialog box, you can define your own symbolic name for a block state:

1. In your block diagram, double-click the desired block. This action opens the block dialog box, containing two or more tab, which includes **State Attributes**.
2. Click the **State Attributes** tab.
3. Enter the symbolic name in the **State name** field. For example, enter the state name `Top_filter`.
4. Click **Apply**.



5 Click **OK**.

The following state initialization code was generated from the example model shown in “Default Block State Naming Convention” on page 19-161, under the following conditions:

- The upper Discrete Filter block has the state name `Top_filter`, and `Auto` storage class (and is therefore stored in the `DWork` vector).
- The lower Discrete Filter block has the state name `Lower_filter`, and storage class `ExportedGlobal`.

`Top_filter` is placed in the `Dwork` vector.

```
/* data type work */
    disc_filt_states_M->Work.dwork = ((void *)
&disc_filt_states_DWork);
    (void)memset((char_T *) &disc_filt_states_DWork, 0,
    sizeof(D_Work_disc_filt_states));
    disc_filt_states_DWork.Top_filter = 0.0;

    /* exported global states */
    Lower_filter = 0.0;
```

Related Examples

- "Control Signals and States in Code by Applying Storage Classes" (Simulink Coder)
- "Continuous Block State Naming in Generated Code" on page 19-158

Initialization of Signal, State, and Parameter Data in the Generated Code

Signal lines, block parameters, and block states in a model can appear in the generated code as data, for example, as global variables. By default, the code initializes this data before execution of the primary algorithm. To match the numerics of simulation in Simulink, the code generator chooses the initial values based on specifications that you make in the model.

By understanding how the generated code initializes data, you can:

- Model a system that reinitializes states during execution, which means that the application can restart the entire system.
- Store initial values in memory as variables, which can persist between execution runs. You can then overwrite these values before starting or restarting a system.
- Generate efficient code by eliminating storage for invariant initial values and by preventing the generation of code that unnecessarily or redundantly initializes data.

For basic information about initializing signals and states in a model, see "Initialize Signals and Discrete States" (Simulink) and "Load State Information" (Simulink).

Static Initialization and Dynamic Initialization

To initialize a data item such as a global variable, an application can use either static or dynamic initialization.

- Static initialization occurs in the same statement that defines (allocates memory for) the variable. The initialization does not occur inside a function definition.

  The code can be more efficient because none of the generated model functions execute initialization statements.
- Dynamic initialization occurs inside a function. For each model or nonvirtual subsystem, the code generator typically creates one or more functions that are dedicated to initialization.

  The generated code or your code can restart a system by calling the initialization function or functions at any time during execution.

Real-World Ground Initialization Requiring Nonzero Bit Pattern

Each data item has a real-world *ground value*. This value can depend on the data type of the item. For example, for a signal whose data type is `double` or `int8`, the real-world ground value is zero. For an enumeration, the ground value is the default enumeration member.

For some kinds of data, to represent a real-world ground value, a computer stores zero in memory (all bits zero). However, for some other data, a computer stores a nonzero value in memory. This data includes, for example:

- Fixed-point data with bias. The code initializes such a data item to the stored integer value that, given the scaling and bias, represents real-world zero.
- An enumeration whose default member maps to an integer value other than zero. For example, if the default member is `High` with an underlying integer value of 3, the code initializes such a data item to `High`.

Initialization of Signal and State Data

In a model, you can control signal and state initial values through block parameters. For example, to set the initial value of a Unit Delay block, you use the **Initial condition** parameter. In some cases, you can also use the `InitialValue` property of a `Simulink.Signal` object. For most of these block parameters, the default value is `0`.

You can also initialize states by using State Writer blocks in Initialize Function subsystems and the **Initial states** model configuration parameter.

By default, the generated code dynamically initializes signal and state data (and other data, such as the model error status) in the generated initialization function. The function, named *`model`*`_initialize` by default, performs signal and state initialization operations in this order:

1. Initializes all of the signal and state data in the default generated structures, such as the DWork structure, to a stored value of zero.
2. Initializes additional signal and state data that are not in the default generated structures to the relevant real-world ground value.

   This initialization applies only to data that meet both of these criteria:

   - The generated code defines (allocates memory for) the data.

- The data use a storage class other than `Auto` (the default storage class) or `SimulinkGlobal`.

For example, the code applies this operation to data items that use the storage class `ExportedGlobal`.

3. Initializes each signal and state to the real-world value that the model specifies, for example, through the **Initial condition** parameter of a Unit Delay block.
4. Initializes each state to the real-world value that you assign by using a State Writer block. The function performs this initialization only if you use an Initialize Function subsystem in the model.
5. Initializes each state to the real-world value that you specify with the configuration parameter **Initial states**.

After this initialization function executes, each data item has the last real-world value that the function assigned. For example, if you use a State Writer block to initialize a block state to `5` while also using the **Initial states** configuration parameter to initialize the same state to `10`, the state ultimately uses `10` as an initial value.

`memset` for Bulk Initialization

To initialize signal or state data items that have contiguous storage to a stored value of zero, the generated code can call `memset` in an initialization function. Data items that have contiguous storage include the DWork structure, arrays, or data items that use a multiword data type.

If your application requires it, you can prevent the generated code from using `memset` for initializing floating-point data to stored zero. See "Use memset to initialize floats and doubles to 0.0" (Simulink).

Tunable Initial Values

You can configure the way that tunable block parameters, such as the **Gain** parameter of a Gain block, appear in the generated code. Most block parameters that set initial values (for example, **Initial condition**) are tunable. For example, the configuration parameter **Default parameter behavior** can determine whether the initial values appear in the generated code as inlined constants or as tunable global data. You can also use parameter objects and storage classes to control the representation of these initial values.

For tunable initial values, in the model initialization function, the right-hand side of the assignment statement is a global variable, structure field, or other data whose value you can change in memory.

To make initial values tunable in the generated code, see “Control Signal and State Initialization in the Generated Code” on page 19-147.

Initialization of Parameter Data

The generated code statically initializes parameter data to the values that you specify in Simulink. For example, these model elements appear in the generated code as parameter data:

- Tunable block parameters, such as the **Gain** parameter of a Gain block, when you set **Default parameter behavior** to `Tunable`. Each parameter appears as a field of a dedicated global structure.
- Some tunable block parameters when you set **Default parameter behavior** to `Inlined`. When the code generator cannot inline the value of a parameter as a literal number, the parameter appears as a field of a dedicated global `const` structure.
- `Simulink.Parameter` objects to which you apply a storage class or custom storage class that has an exported data scope. For example, the built-in storage class `ExportedGlobal` has an exported data scope, but the storage class `ImportedExtern` does not.

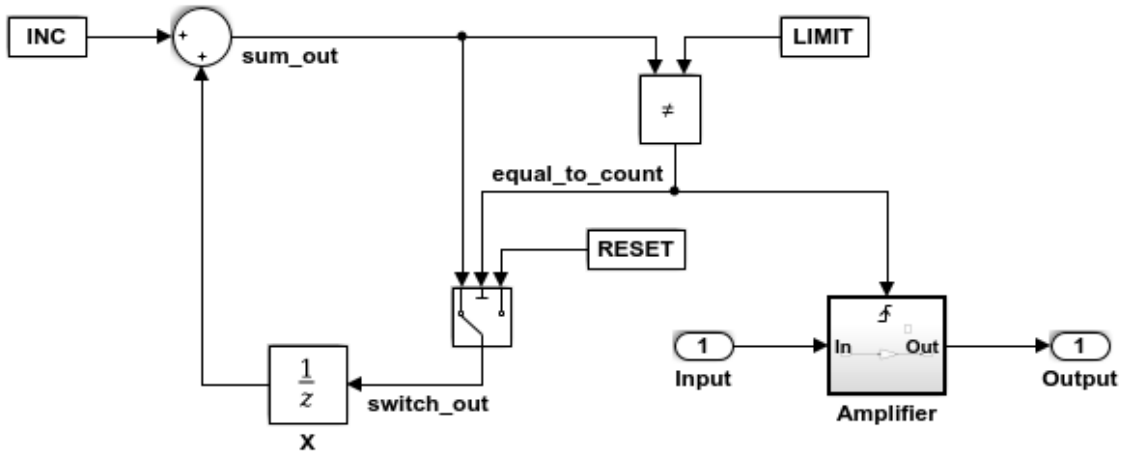
Data Initialization in the Generated Code

This example shows how the generated code initializes signal, state, and parameter data.

Explore Example Model

Open the example model, `rtwdemo_rtwintro`.

```
open_system('rtwdemo_rtwintro')
```



In the Unit Delay block, the **Initial condition** parameter is set to `0`, the default, which means the initial value of the block state is zero.

Set **Initial condition** to a nonzero number, for example, `5`.

```
set_param('rtwdemo_rtwintro/X','InitialCondition','5')
```

Inspect the **State Attributes** tab. The block state is named `X`.

Set **Code generation storage class** to `ExportedGlobal`. When you choose this setting, the block state appears in the generated code as a separate global variable.

```
set_param('rtwdemo_rtwintro/X','StateStorageClass','ExportedGlobal')
```

In the model, open the `Amplifier` subsystem, which is a triggered subsystem.

In the subsystem, in the Outport block, inspect the **Initial output** parameter. The value is `0`, the default. This subsystem output requires an initial value because the subsystem executes conditionally. Leave the initial value at the default.

In the model, clear **Configuration Parameters > All Parameters > Signal storage reuse**. When you clear this setting, signal lines appear in the generated code as fields of a generated structure whose purpose is to store signal data. This representation of the signals makes it easier to see how the code generator initializes data.

```
set_param('rtwdemo_rtwintro','OptimizeBlockIOStorage','off')
```

In the model, inspect the configuration parameter **Configuration Parameters > Optimization > Signals and Parameters > Default parameter behavior**. The configuration parameter is set to `Inlined`, which means block parameters, including initial values, appear in the generated code as inlined literals or as `const` data.

In the model, inspect the Constant block labeled `INC`. The **Constant value** parameter of the block is set to `INC`. `INC` is a MATLAB variable in the base workspace.

At the command prompt, convert `INC` to a `Simulink.Parameter` object and apply the storage class `ExportedGlobal`. With these settings, the generated code defines `INC` as a global variable. Concerning initialization, `INC` is an item of parameter data.

```
INC = Simulink.Parameter(INC);
INC.StorageClass = 'ExportedGlobal';
```

Generate and Inspect Code

Generate code from the model.

```
rtwbuild('rtwdemo_rtwintro');
```

```
### Starting build procedure for model: rtwdemo_rtwintro
### Successful completion of build procedure for model: rtwdemo_rtwintro
```

In the generated file `rtwdemo_rtwintro.c`, outside the definition of any function, the code statically initializes the parameter data `INC`. The value of `INC` is `1`.

```
file = fullfile('rtwdemo_rtwintro_grt_rtw','rtwdemo_rtwintro.c');
rtwdemodbtype(file,'/* Exported block parameters */','uint8_T INC = 1U;',1,1)
```

```
/* Exported block parameters */
uint8_T INC = 1U;                      /* Variable: INC
```

In the same file, inspect the definition of the `rtwdemo_rtwintro_initialize` function. First, the function uses `memset` to initialize all of the internal signals in the model to a stored value of `0`.

```
rtwdemodbtype(file,'/* block I/O */','sizeof(B_rtwdemo_rtwintro_T));',1,1)
```

```
  /* block I/O */
  (void) memset(((void *) &rtwdemo_rtwintro_B), 0,
                sizeof(B_rtwdemo_rtwintro_T));
```

The function then initializes the Unit Delay state, `X`, to the appropriate ground value. In this case, the ground value is zero.

```
rtwdemodbtype(file,'/* exported global states */','X = 0U;',1,1)
```

```
  /* exported global states */
  X = 0U;
```

The function also initializes other data, including the root-level inputs and outputs (Inport and Outport blocks), to the appropriate ground values.

```
rtwdemodbtype(file,'/* external inputs */','Amplifier_Trig_ZCE = POS_ZCSIG;',1,1)
```

```
  /* external inputs */
  rtwdemo_rtwintro_U.Input = 0;

  /* external outputs */
  rtwdemo_rtwintro_Y.Output = 0;
  rtwdemo_rtwintro_PrevZCX.Amplifier_Trig_ZCE = POS_ZCSIG;
```

The function then initializes `X` to the value that you specified in the **Initial condition** block parameter.

```
rtwdemodbtype(file,'/* InitializeConditions for UnitDelay: ''<Root>/X'' */',...
    'X = 5U;',1,1)
```

```
  /* InitializeConditions for UnitDelay: '<Root>/X' */
  X = 5U;
```

Finally, the function initializes the output of the `Amplifier` subsystem.

```
rtwdemodbtype(file,'SystemInitialize for Triggered SubSystem',...
    'End of SystemInitialize for SubSystem',1,1)
```

```
/* SystemInitialize for Triggered SubSystem: '<Root>/Amplifier' */
/* SystemInitialize for Outport: '<Root>/Output' incorporates:
 *  SystemInitialize for Outport: '<S1>/Out'
 */
rtwdemo_rtwintro_Y.Output = 0;
```

Inspect Difference Between Stored Value and Real-World Value

For some data, even if the real-world initial value is zero, a computer stores a nonzero value in memory. To observe this difference, apply a slope-bias fixed-point data type to the block state, `X`. To run this example, you must have Fixed-Point Designer™.

In the Switch block that feeds the Unit Delay block, on the **Signal Attributes** tab, set **Output data type** to `fixdt(1,16,1,3)`. This expression represents a fixed-point data type with a slope of `1` and a bias of `3`.

```
set_param('rtwdemo_rtwintro/Switch','OutDataTypeStr',...
    'fixdt(1,16,1,3)')
```

In the Sum block, on the **Signal Attributes** tab, clear **Require all inputs to have the same data type**.

```
set_param('rtwdemo_rtwintro/Sum','InputSameDT','off')
```

To prevent compilation errors on different platforms, select the model configuration parameter **Generate code only**. This setting causes the model to generate only code.

```
set_param('rtwdemo_rtwintro','GenCodeOnly','on')
```

Generate code from the model.

```
rtwbuild('rtwdemo_rtwintro')
```

```
### Starting build procedure for model: rtwdemo_rtwintro
### Successful completion of code generation for model: rtwdemo_rtwintro
```

The model initialization function first initializes `X` to the appropriate real-world ground value, `0`. Due to the slope-bias data type, which has bias `3`, this real-world value corresponds to a stored value of `-3`.

```
rtwdemodbtype(file,'/* exported global states */','X = -3;',1,1)
```

```
  /* exported global states */
```

```
X = -3;
```

The function then initializes `X` to the real-world initial value that you specified in the **Initial condition** block parameter, `5`. This real-world value corresponds to a stored value of `2`.

```
rtwdemodbtype(file,'/* InitializeConditions for UnitDelay: ''<Root>/X'' */',...
    'X = 2;',1,1)
```

```
/* InitializeConditions for UnitDelay: '<Root>/X' */
X = 2;
```

Modeling Goals

Goal	More Information
Explicitly model initialization behavior by using blocks	You can explicitly model initialization and reset behavior by using Initialize Function and Reset Function subsystems. In the subsystems, use State Writer blocks to calculate and assign an initial value for a state dynamically. The corresponding code appears in the model initialization function. For more information, see “Generate Code That Responds to Initialize, Reset, and Terminate Events” on page 9-2.
Prevent generation of code that explicitly initializes data to zero	If your application environment already initializes global variables to zero, for more efficient code, you can prevent the generation of statements that explicitly initialize global variables to zero. This optimization applies only to signals and states whose initial values are stored in memory as zero. For example, the code generator does not apply the optimization to: • Data for which you specify a nonzero initial value by using a block parameter. • Data whose real-world initial value is zero but whose corresponding stored value is not zero. • Enumerated data whose default member maps to a nonzero integer. The optimization requires Embedded Coder. For more information, see “Remove Initialization Code” on page 56-3.

Goal	More Information
Generate code that imports data from your handwritten code	You can generate code that reuses (imports) data that your handwritten code defines. For example, you can apply the storage class `ImportedExtern` to a signal line, block state, or parameter object. For imported data: • The generated code does not initialize parameter data. Your code must initialize imported parameter data. • The generated initialization functions dynamically initialize signal and state data. Your code does not need to initialize imported signal or state data. Unlike data that the generated code allocates, the code does not initialize imported signal or state data to a stored value of zero. Instead, the code immediately initializes the data to the real-world value that you specify in Simulink. To prevent initialization of signal, state, or parameter data by the generated code, you can create a custom storage class and apply it to the data items. In the Custom Storage Class Designer, in the definition of your custom storage class, set **Data initialization** to `None`. For more information about integrating the generated code with external code, see “What Is External Code Integration?” on page 39-3. For information about custom storage classes, see “Introduction to Custom Storage Classes” on page 23-2.

See Also

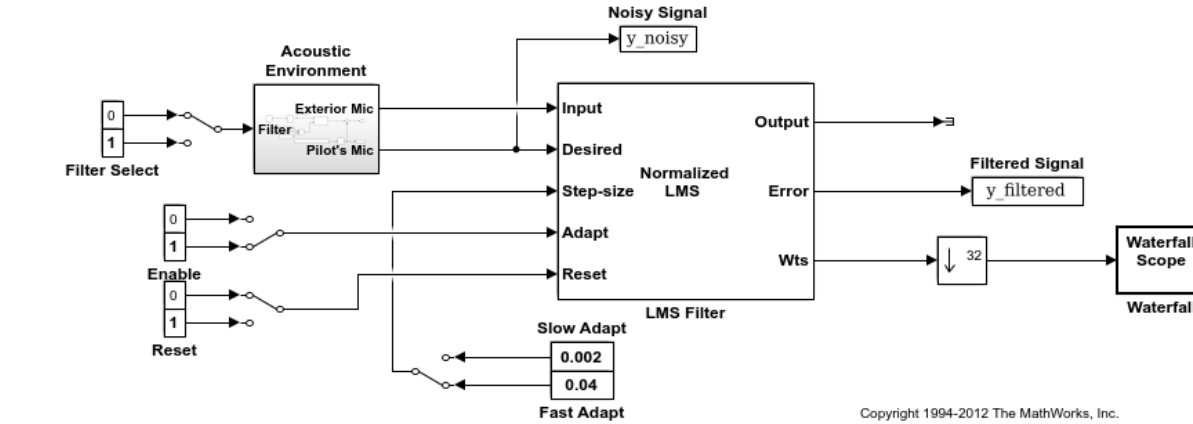
`model_initialize`

Related Examples

- “Signal Representation in Generated Code” on page 19-112

Signal Processing with Fixed-Point Data

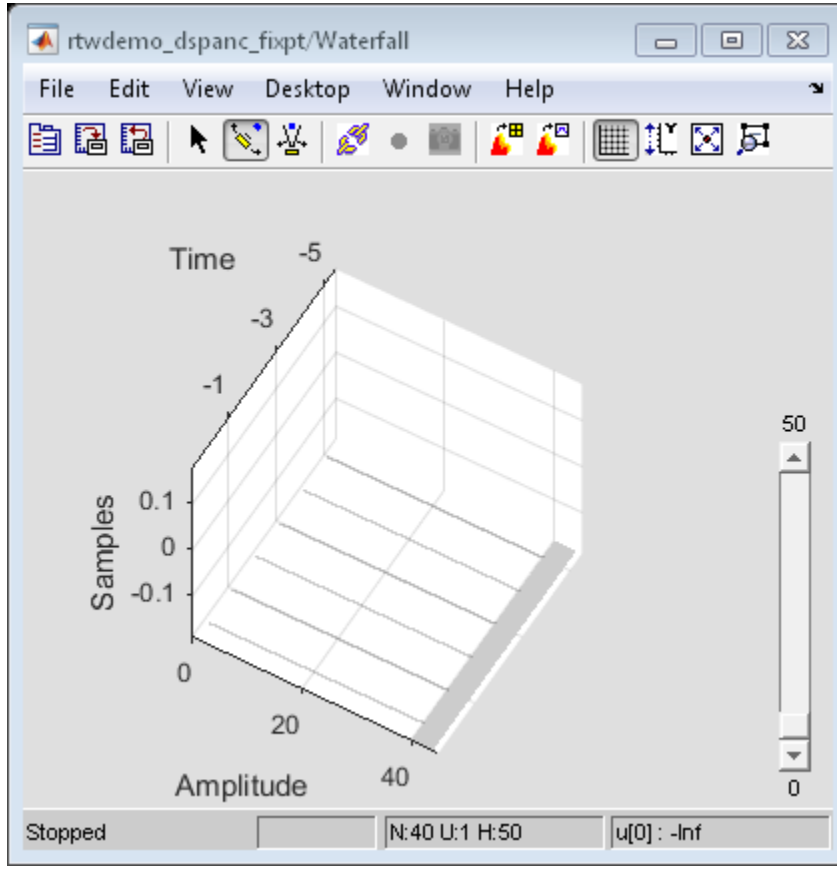
This model shows a fixed-point version of an acoustic noise canceller.



Generate Code Using Simulink Coder (double-click)

Generate Code Using Embedded Coder (double-click)

Acoustic Noise Canceler (Fixed-Point Version). See the Signal Processing Blockset documentation for details on the dspanc example.



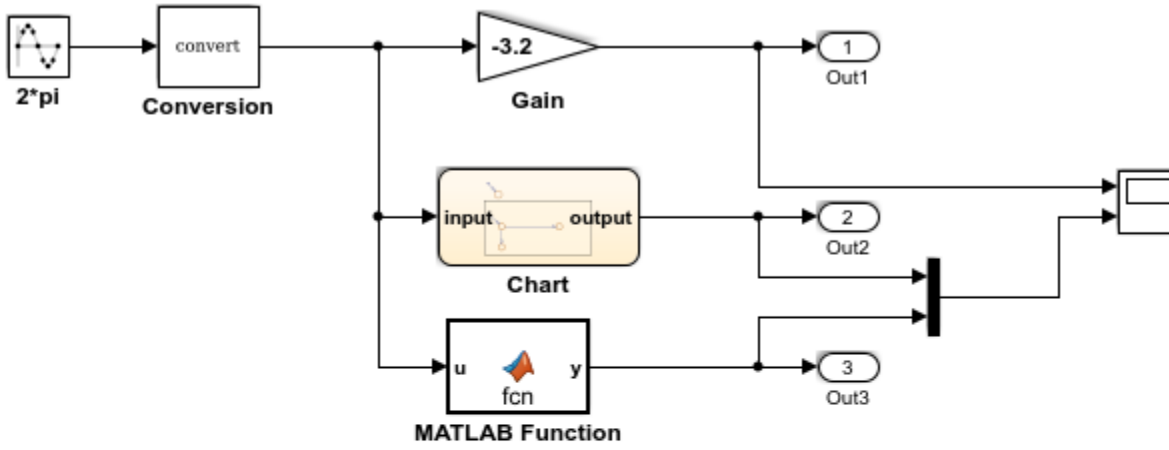
More About

- "Fixed Point" (Simulink)
- "Acoustic Noise Cancellation (LMS)" (DSP System Toolbox)

Optimize Generated Code Using Fixed-Point Data with Simulink®, Stateflow®, and MATLAB®

This model shows fixed-point code generation in Simulink®, Stateflow®, and MATLAB®.

```
open_system ('rtwdemo_fixpt1');
```



This introductory model shows fixed-point code generation in Simulink, Stateflow, and MATLAB. To generate and inspect the code, double-click the blue buttons below. An HTML report detailing the code is displayed automatically.

Generate Code Using Simulink Coder (double-click)

Generate Code Using Embedded Coder (double-click)

More About

- "Fixed Point" (Simulink)

Declare Workspace Variables as Tunable Parameters Using the Model Parameter Configuration Dialog Box

You can use the Model Parameter Configuration dialog box to declare numeric MATLAB variables in the base workspace as tunable parameters. You can select code generation options, such as storage class, for each tunable parameter.

However, it is a best practice to instead use parameter objects to declare tunable parameters. Do not use the Model Parameter Configuration dialog box to select parameter objects in the base workspace. To use parameter objects, instead of the Model Parameter Configuration dialog box, to declare tunable parameters, see "Override Default Parameter Behavior by Creating Global Variables in the Generated Code" on page 19-49.

Note You cannot use the Model Parameter Configuration dialog box to declare tunable parameters for a referenced model. Use `Simulink.Parameter` objects instead.

Declare Existing Workspace Variables as Tunable Parameters

Use the Model Parameter Configuration dialog box to declare existing workspace variables as tunable parameters for a model.

1. In the Configuration Parameters dialog box, on the **Optimization > Signals and Parameters** pane, click **Configure**.
2. In the Model Parameter Configuration dialog box, under **Source list**, select a method to populate the list of available workspace variables.
   - Select `MATLAB workspace` to view all of the numeric variables that are defined in the base workspace.
   - Select `Referenced workspace variables` to view only the numeric variables in the base workspace that the model uses. Selecting this option begins a diagram update and a search for used variables, which can take time for a large model.
3. In the Model Parameter Configuration dialog box, under **Source list**, select one or more workspace variables.
4. Click **Add to table**. The variables appear as tunable parameters under **Global (tunable) parameters**, and appear in italic font under **Source list**.

5 Optionally, select a parameter under **Global (tunable) parameters**, and adjust the code generation settings for the parameter. For more information about adjusting the code generation options for tunable parameters, see "Set Tunable Parameter Code Generation Options" on page 19-179

6 Click **OK** to apply your selection of tunable parameters and close the dialog box.

Declare New Tunable Parameters

Use the Model Parameter Configuration dialog box to declare new tunable parameters. You can use this technique to declare the names of tunable parameters, and to adjust their code generation settings, before you create the corresponding workspace variables.

1 In the Configuration Parameters dialog box, on the **Optimization > Signals and Parameters** pane, click **Configure**.

2 In the Model Parameter Configuration dialog box, under **Global (tunable) parameters**, click **New**.

3 Under the **Name** column, specify a name for the new tunable parameter.

4 Optionally, adjust the code generation settings for the new parameter. For more information about adjusting the code generation options for tunable parameters, see "Set Tunable Parameter Code Generation Options" on page 19-179

5 Click **OK** to apply your changes and close the dialog box.

Set Tunable Parameter Code Generation Options

To set the properties of tunable parameters listed under **Global (tunable) parameters** in the Model Parameter Configuration dialog box, select a parameter and specify a storage class and, optionally, a storage type qualifier.

Property	Description
Storage class	Select one of the following to use for code generation: • `SimulinkGlobal (Auto)` • `ExportedGlobal` • `ImportedExtern` • `ImportedExternPointer` For more information about tunable parameter storage classes, see "Override Default Parameter Behavior by

Property	Description
	Creating Global Variables in the Generated Code" on page 19-49.
Storage type qualifier	For variables with a storage class other than `SimulinkGlobal (Auto)`, you can add a qualifier (such as `const` or `volatile`) to the generated storage declaration. To do so, you can select a predefined qualifier from the list, or add qualifiers not in the list by typing them in. The code generator does not check the storage type qualifier for validity, and includes the qualifier text in the generated code without checking syntax .

Programmatically Declare Workspace Variables as Tunable Parameters

Tune Parameters from the Command Line

When parameters are MATLAB workspace variables, the Model Parameter Configuration dialog box is the recommended way to see or set the properties of tunable parameters. In addition to that dialog box, you can also use MATLAB `get_param` and `set_param` commands.

Note You can also use `Simulink.Parameter` objects for tunable parameters. See "Block Parameter Representation in the Generated Code" (Simulink Coder) for details.

The following commands return the tunable parameters and corresponding properties:

- `get_param(gcs,'TunableVars')`
- `get_param(gcs,'TunableVarsStorageClass')`
- `get_param(gcs,'TunableVarsTypeQualifier')`

The following commands declare tunable parameters or set corresponding properties:

- `set_param(gcs,'TunableVars',str)`

  The argument `str` (character vector) is a comma-separated list of variable names.
- `set_param(gcs,'TunableVarsStorageClass',str)`

The argument `str` (character vector) is a comma-separated list of storage class settings.

The valid storage class settings are

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

- `set_param(gcs,'TunableVarsTypeQualifier',str)`

  The argument `str` (character vector) is a comma-separated list of storage type qualifiers.

The following example declares the variable `k1` to be tunable, with storage class `ExportedGlobal` and type qualifier `const`. The number of variables and number of specified storage class settings must match. If you specify multiple variables and storage class settings, separate them with a comma.

```
set_param(gcs,'TunableVars','k1')
set_param(gcs,'TunableVarsStorageClass','ExportedGlobal')
set_param(gcs,'TunableVarsTypeQualifier','const')
```

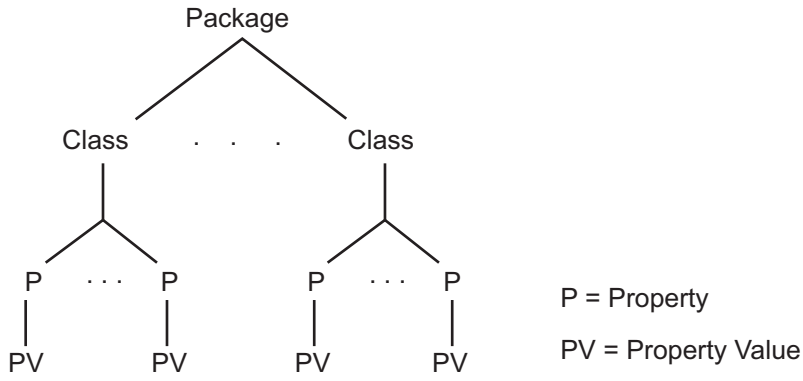
20

Data Definition and Declaration Management in Embedded Coder

Overview of Data Objects

Data objects include the parameters and signals that the source code uses, and a description of their properties. Data objects appear in the middle pane of the Model Explorer. They also appear in the MATLAB workspace. You can control the property values for each data object, thereby determining how each parameter and signal is defined and declared in generated code.

Simulink uses a hierarchy of terms that are drawn from object-oriented programming. For details, see "Data Objects" (Simulink). The sketch below summarizes this hierarchy.



You can use the `Simulink.Parameter` class to declare a data object for a parameter, where `Simulink` is the package name and Parameter is the class name. Likewise, an instance of a `Simulink.Signal` class, creates a data object for a signal. Signal data objects have a different set of properties than a parameter data objects. When you create a data object, you specify a value for each of the properties, which defines that object. For more information, see Simulink.Parameter class (Simulink) and Simulink.Signal (Simulink).

Related Examples

- "Create Data Objects for Code Generation with Data Object Wizard" on page 24-2
- "Place Global Data Declarations and Definitions in Separate Files" on page 20-3

Place Global Data Declarations and Definitions in Separate Files

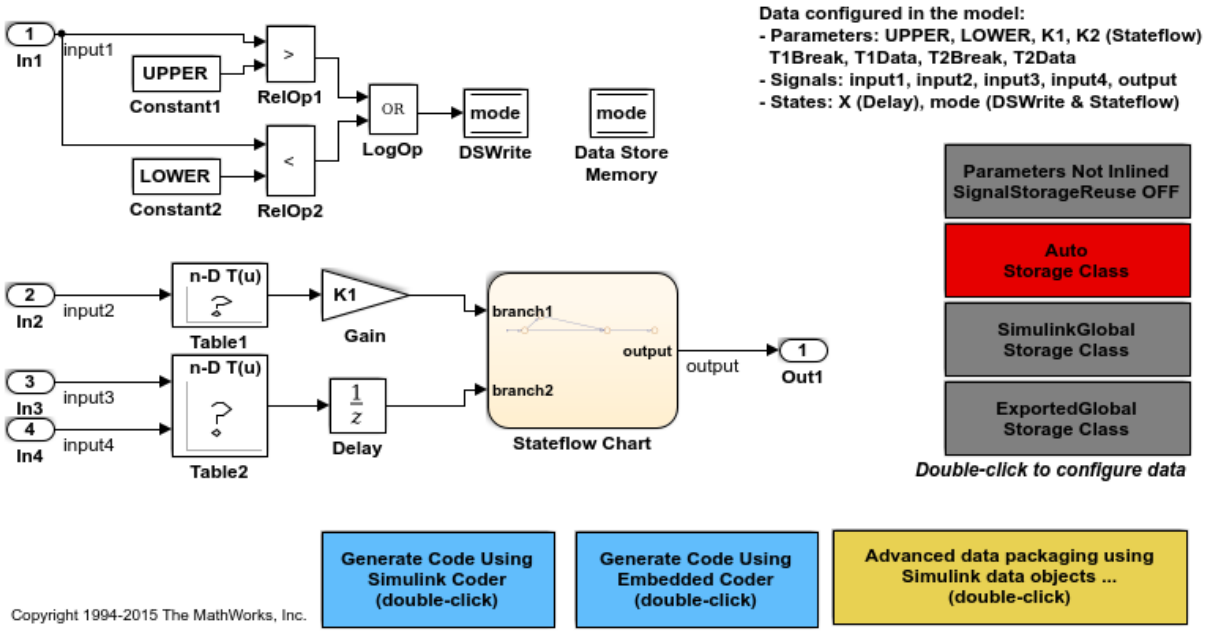
These examples show how to control the file placement of data items (signals, parameters, and states) to which you apply storage classes and custom storage classes.

Place Multiple Data Items in Single File by Default

By default, the declarations and definitions generated for individual data items typically appear in the model source file. This example shows how to modularize the code by placing these global data in a separate file.

Open the example model rtwdemo_basicsc.

```
open_system('rtwdemo_basicsc')
```



The model creates numeric variables in the base workspace. Blocks in the model use these variables to set parameter values (such as the **Gain** parameter of a Gain block). Some of the signals and block states in the model have explicit names, such as `input1`.

1. Select **Code > Data Objects > Data Object Wizard**.
2. In the Data Object Wizard, click **Find**. The Data Object Wizard proposes the creation of `Simulink.Parameter` objects to replace the workspace variables and the creation of `Simulink.Signal` objects to represent the named signals and states.
3. Click **Select All** and **Create**. The Data Object Wizard creates the data objects in the base workspace. You can use these objects to specify code generation settings for the corresponding signals, parameters, and states in the model.
4. In the Model Explorer **Model Hierarchy** pane, select **Base Workspace**.
5. In the **Contents** pane, set **Column View** to `Storage Class`.
6. For all of the data objects, use the **StorageClass** column to apply the custom storage class `Default`. With this custom storage class, each data object appears in the generated code as a separate global variable.
7. In the model, set **Configuration Parameters > Code Generation > System target file** to `ert.tlc`. With this setting, the code generator honors custom storage classes such as `Default`.
8. Specify that global data items be defined in a separate file. Set **Configuration Parameters > Code Generation > Code Placement > Data definition** to `Data defined in a single separate source file`. Accept the default for **Data definition filename**, `global.c`.
9. Specify that data be declared in a separate file. Set **Data declaration** to `Data declared in a single separate header file` and accept the default for **Data declaration filename**, `global.h`. Then, click **Apply**.
10. Generate code from the model. Notice that the code generation report lists `global.c` and `global.h` files.
11. Inspect the code generation report. Notice that:

- The global data are defined and, for parameters, initialized in `global.c`.
- The file `rtwdemo_basicsc.c` includes `(#include) rtwdemo_basicsc.h`.
- The file `rtwdemo_basicsc.h` includes `global.h`.

Place Each Data Item in Individual File

In the previous example, you place global data in a separate definition file and a declaration file. You name the files `global.c` and `global.h`. You can override this specification and place each individual data item in its own file. In this example, move the `output` signal to a file named `outputsig.c`. Keep the other data defined in `global.c`.

1. In your current folder, delete the `slprj` subfolder.
2. In the Model Explorer, display the base workspace and select the `output` signal object. The **Simulink.Signal** properties appear in the right pane.
3. In the **Code generation options** section, set **Storage class** to `ExportToFile`. Set **HeaderFile** to `outputsig.h` and **DefinitionFile** to `outputsig.c`. Click **Apply**.
4. Generate code from the model. The code generation report still lists `global.c` and `global.h`, but adds `outputsig.c` and `outputsig.h`.
5. Inspect the new files. The `output` signal is defined in `outputsig.c`. Other data are still defined in `global.c`.

See Also

Simulink.Parameter | `Simulink.Signal`

Related Examples

- “Manage Placement of Data Definitions and Declarations” on page 36-100
- “Control Signals and States in Code by Applying Storage Classes” (Simulink Coder)
- “Block Parameter Representation in the Generated Code” (Simulink Coder)
- “Data Objects” (Simulink)

Data Types in Embedded Coder

What Are User-Defined Data Types?

User-defined data types are objects of the following data type classes.

- `Simulink.AliasType`
- `Simulink.NumericType`

You can apply user-defined data types to achieve the following objectives in the generated code.

- Specify custom data type names for individual block parameters and signals by creating aliases of the built-in Simulink types. You can configure the aliases to appear in the block diagram and in generated code. For more information, see "Create Data Type Alias in the Generated Code" on page 21-12.
- Map your own data type definitions to the built-in data types, and specify that your data types are to be used in generated code. For more information, see "Data Type Replacement" on page 21-36.
- Optionally, generate `#include` directives to import header files that contain your data type definitions. This technique allows you to 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. For `Simulink.NumericType` objects whose `IsAlias` property is `false` or `0`, the name of the functionally equivalent built-in or fixed-point Simulink data type is used instead.

- To create your own data type as an alias of a built-in data type and share the type between data items in a model, you can use a `Simulink.AliasType` object or a `Simulink.NumericType` object. For a `Simulink.NumericType` object, set the `IsAlias` property to `true`.

  To create an alias of an enumerated data type, you must use a `Simulink.AliasType` object.
- To share a numeric data type such as `single`, `int16`, or a fixed-point data type without renaming the type, use a `Simulink.NumericType` object and set `IsAlias` to `false` (the default).

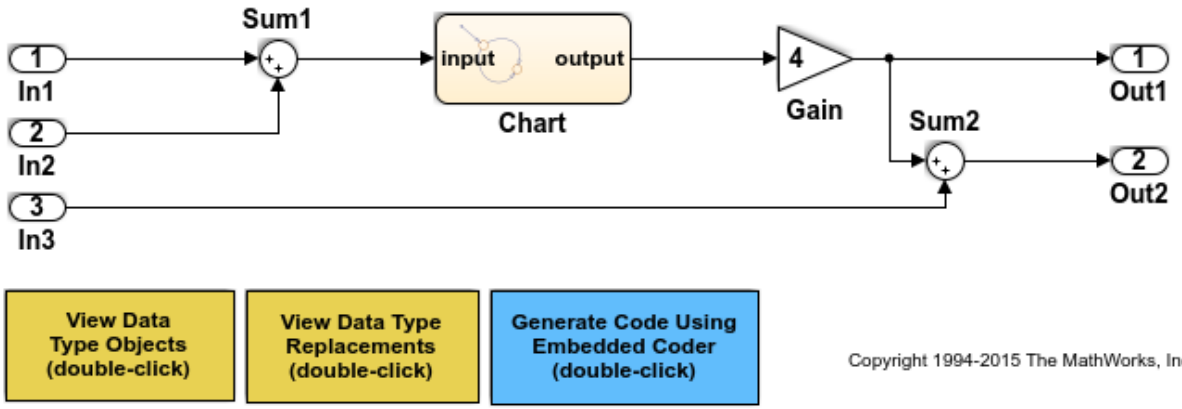
Define Abstract Numeric Types and Rename Types

This model shows user-defined types, consisting of numeric and alias types. Numeric types allow you to define abstract numeric types, which is particularly useful for fixed-point types. Alias types allow you to rename types, which allows you create a relationship for types.

Explore Example Model

Open the example model.

```
open_system('rtwdemo_udt')
```



Key Features of User-Defined Types

- Displayed and propagated on signal lines
- Used to parameterize a model by type (e.g., In1 specifies its **Output data type** as `ENG_SPEED`)
- Types with a common ancestor can be mixed, whereby the common ancestor is propagated (e.g., output of Sum1)
- Intrinsically supported by the Simulink Model Explorer
- Include an optional header file attribute that is ideal for importing legacy types (ignored for GRT targets)
- Types used in the generated code (ignored for GRT targets)

Instructions

1. Inspect the user-defined types in the Model Explorer by double-clicking the first yellow button below.
2. Inspect the replacement data type mapping by double-clicking the second yellow button below.
3. Compile the diagram to display the types in this model (Simulation > Update Diagram or **Ctrl+D**).
4. Generate code with the blue button below and inspect model files to see how user-defined types appear in the generated code.
5. Modify the attributes of `ENG_SPEED` and `ENG_SPEED_OFFSET` and repeat steps 1-4.

Notes

- User-defined types are a feature of Simulink that facilitate parameterization of the data types in a model. Embedded Coder preserves alias data type names (e.g., `ENG_SPEED`) in the generated code, whereas Simulink Coder implements user-define types as their base type (e.g., `real32_T`).
- Embedded Coder also enables you to replace the built-in data types with user-defined data types in the generated code.

Rename Data Type Object

To rename a data type object after you create it (for example, to rename an alias when coding standards change or when you encounter a naming conflict), you can allow Simulink to rename the object and correct all of the references to the object that appear in a model or models. In the Model Explorer, right-click the variable and select **Rename All**. For more information, see "Rename Variable" (Simulink).

Enumerations and Structures

To generate Simulink representations of custom `struct` and `enum` data types that your C code defines, use the `Simulink.importExternalCTypes` function to create corresponding `Simulink.Bus` objects and enumeration classes.

Related Examples

- "Unit Specification in Simulink Models" (Simulink)

- “Create Data Type Alias in the Generated Code” on page 21-12
- “Data Type Replacement” on page 21-36
- “Group Signals into Structures in the Generated Code Using Buses” on page 19-139
- “Data Objects” (Simulink)

Control File Placement of User-Defined Types

In this section...

When you use data type objects such as `Simulink.AliasType` to specify data types for signals and block parameters, the code generated from the model defines the types with `typedef` statements. To ease integration of the generated code with other existing code, you can control the file placement of the `typedef` statements by adjusting the properties of the objects.

Data Scope and Header File

To control the file placement of a `typedef` statement in generated code, set the `DataScope` and `HeaderFile` properties of the data type object according to the table.

- *`typename`* is the name of the custom data type.
- *`filename`* is the name of a header file.
- *`model`* is the name of the model.

Goal	Specify `DataScope` as	Specify `HeaderFile` as
Export type definition to *`model`*`_types.h`	`Auto`	Empty
Import type definition from a header file that you create, *`filename`*`.h`	`Auto` or `Imported`	*`filename`*`.h`
Export type definition to a generated header file, *`filename`*`.h`	`Exported`	*`filename`*`.h`
Import type definition from a header file that you create, *`typename`*`.h`	`Imported`	Empty
Export type definition to a generated header file, *`typename`*`.h`	`Exported`	Empty

When you import a data type definition, the generated model code creates an `#include` directive for your header file in place of a `typedef` statement. You must supply the header file that contains the `typedef` statement.

By default, the generated `#include` directives use the preprocessor delimiter `"` instead of `<` and `>`. To generate the directive `#include <myTypes.h>`, specify the `HeaderFile` property as `<myTypes.h>`.

Data Type Replacement

If you use Data Type Replacement to replace a built-in Simulink data type with your own data type in generated code, `typedef` statements and `#include` directives appear in `rtwtypes.h` instead of *`model`*`_types.h`.

Macro Guards

When you export one or more data type definitions to a generated header file, the file contains a file-level macro guard of the form `RTW_HEADER_`*`filename`*`_h`.

Suppose you use several `Simulink.AliasType` objects: `mySingleAlias`, `myDoubleAlias`, and `myIntAlias` with these properties:

- `DataScope` set to `Exported`
- `HeaderFile` set to `myTypes.h`

When you generate code, the guarded file `myTypes.h` contains the `typedef` statements:

```
#ifndef RTW_HEADER_myTypes_h_
#define RTW_HEADER_myTypes_h_
#include "rtwtypes.h"

typedef real_T myDoubleAlias;
typedef real32_T mySingleAlias;
typedef int16_T myIntAlias;

#endif
```

When you export data type definitions to *`model`*`_types.h`, the file contains a macro guard of the form `_DEFINED_TYPEDEF_FOR_`*`typename`*`_` for each `typedef` statement. Suppose you use a `Simulink.AliasType` object `mySingleAlias` with these properties:

- `DataScope` set to `Auto`

- `HeaderFile` not specified

When you generate code, the file *`model`*`_types.h` contains the guarded `typedef` statement:

```
#ifndef _DEFINED_TYPEDEF_FOR_mySingleAlias_
#define _DEFINED_TYPEDEF_FOR_mySingleAlias_

typedef real32_T mySingleAlias;

#endif
```

See Also

`Simulink.AliasType` | Simulink.Bus | `Simulink.NumericType`

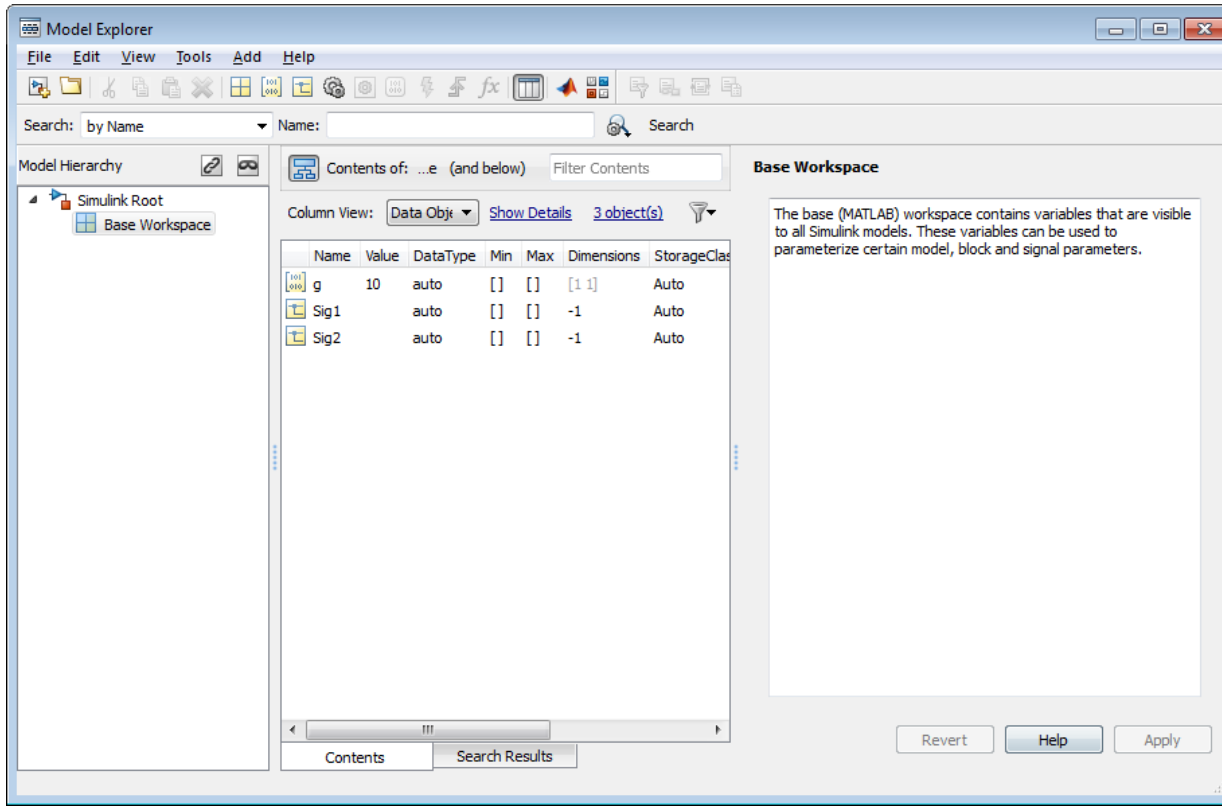
Related Examples

- “Create Data Type Alias in the Generated Code” on page 21-12
- “Data Type Replacement” on page 21-36
- “What Are User-Defined Data Types?” on page 21-2

Create and Apply User-Defined Data Types

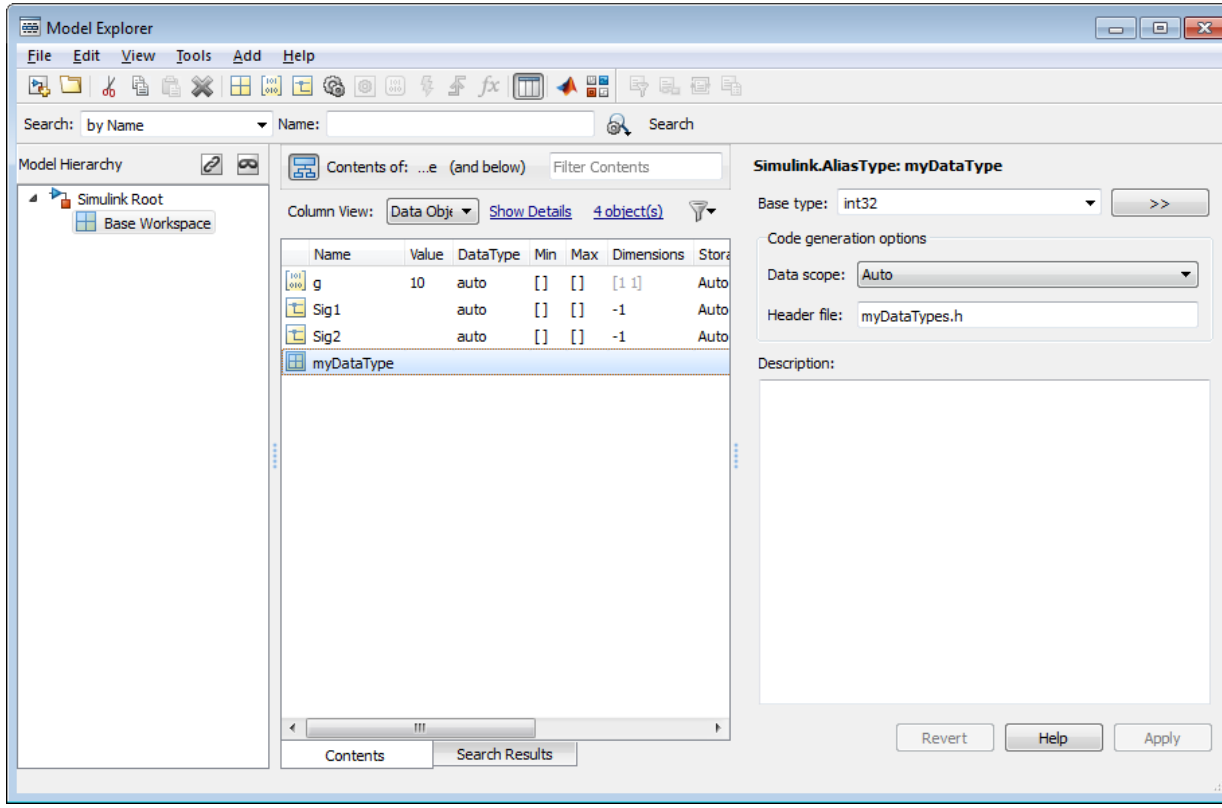
This example shows how to create user-defined data types and specify them for data objects.

1 Open the Model Explorer and create `Simulink.Signal` and `Simulink.Parameter` objects in the base workspace.



2 Click **Add** > **Simulink.AliasType** to create a data type object.

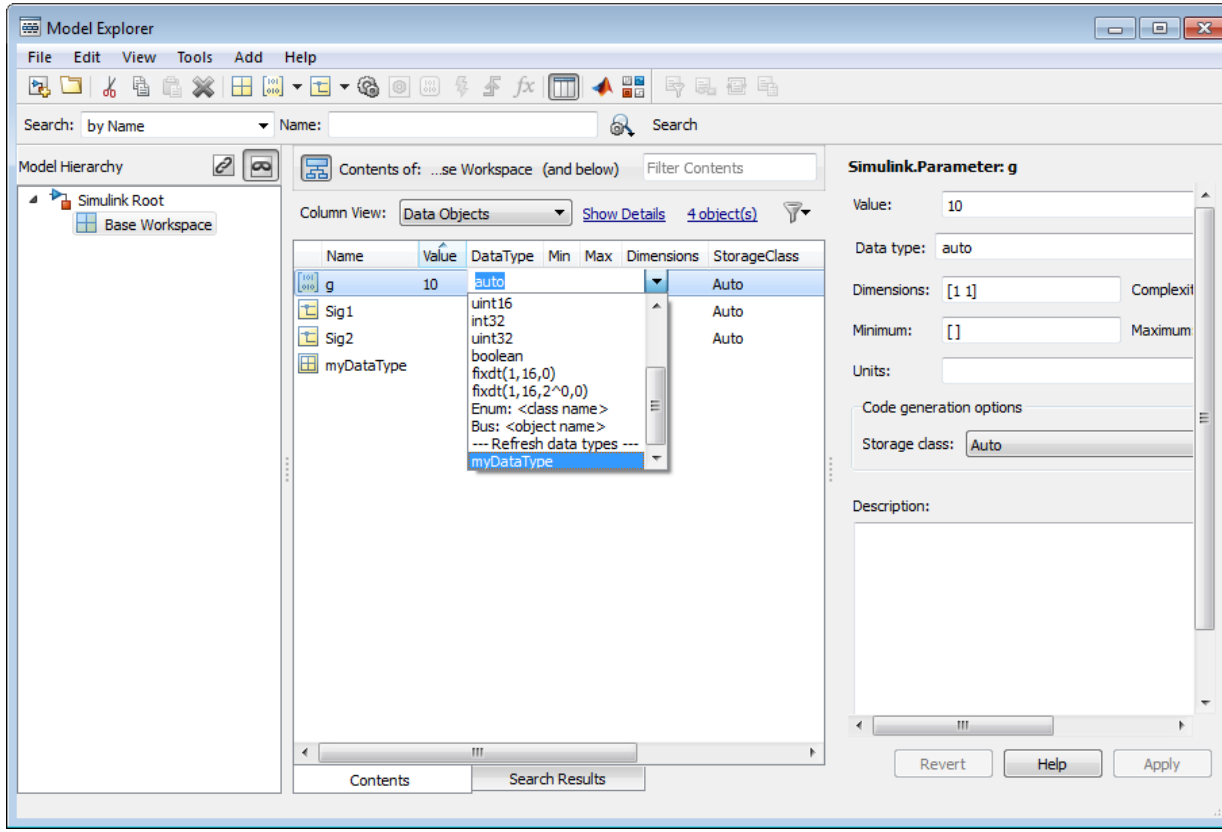
3 Name the object and set its **Base type** to `int32` and **Header file** to `myDataTypes.h`.



4 Select the data object for which you want to specify the user-defined data type. Click its **Data Type** field and from the drop down select **Refresh data types**.

   This action updates the data type list with the user-defined data type you created.

5 Select the user-defined data type.



See Also

`Simulink.AliasType` | `Simulink.importExternalCTypes`

Related Examples

- “Create Data Type Alias in the Generated Code” on page 21-12
- “Data Objects” (Simulink)
- “Parameter Data Types in the Generated Code” on page 19-79
- “Data Type Replacement” on page 21-36
- “Create a Named Fixed-Point Data Type in the Generated Code” on page 21-18

Create Data Type Alias in the Generated Code

You can create your own data type in code that a model generates by using an alias of an existing type. You can use the alias to specify parameter and signal data types throughout a model diagram and in generated code.

You can use an alias for the built-in Simulink data types, custom enumerated types that you create, and fixed-point data types that you create. To create a data type alias, you use an object of the class `Simulink.AliasType`.

You can also rename a built-in Simulink type in code generated from a model without using a data type alias in the model diagram. For more information, see "Data Type Replacement".

Export Type Definition

When you integrate code generated from a model with code from other sources, your model code can create an exported `typedef` statement. Therefore, all of the integrated code can use the type. This example shows how to export the definition of a data type to a generated header file.

Create a `Simulink.AliasType` object named `mySingleAlias` that acts as an alias for the built-in data type `single`.

```
mySingleAlias = Simulink.AliasType('single')
```

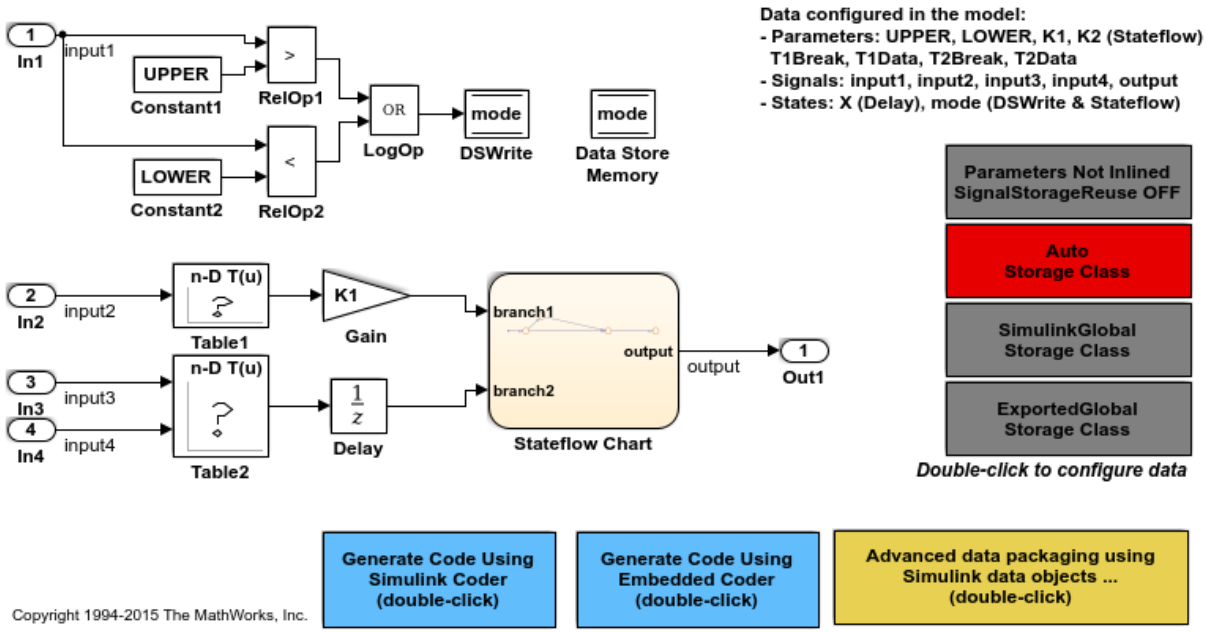
```
mySingleAlias =

  AliasType with properties:

    Description: ''
      DataScope: 'Auto'
     HeaderFile: ''
       BaseType: 'single'
```

Configure the object to export its definition to a header file called `myHdrFile.h`.

```
mySingleAlias.DataScope = 'Exported';
mySingleAlias.HeaderFile = 'myHdrFile.h';
```

Open the model `rtwdemo_basicsc`.

```
open_system('rtwdemo_basicsc')
```



Inspect the parameters of the Inport block labeled `In1`. On the **Signal Attributes** tab, **Data type** is set to `single`.

Set **Data type** to the alias `mySingleAlias`.

```
set_param('rtwdemo_basicsc/In1','OutDataTypeStr','mySingleAlias')
```

In the model, set **Configuration Parameters > Code Generation > System target file** to `ert.tlc`. With this setting, the code generator honors data type aliases such as `mySingleAlias`.

```
set_param('rtwdemo_basicsc','SystemTargetFile','ert.tlc')
```

Generate code from the model.

```
rtwbuild('rtwdemo_basicsc')
```

```
### Starting build procedure for model: rtwdemo_basicsc
### Successful completion of build procedure for model: rtwdemo_basicsc
```

In the code generation report, view the file `rtwdemo_basicsc_types.h`. The code creates a `#include` directive for the generated file `myHdrFile.h`.

```
file = fullfile('rtwdemo_basicsc_ert_rtw','rtwdemo_basicsc_types.h');
rtwdemodbtype(file,'#include "myHdrFile.h"',...
    '#include "myHdrFile.h"',1,1)
```

```
#include "myHdrFile.h"
```

View the file `myHdrFile.h`. The code uses the identifier `mySingleAlias` as an alias for the data type `real32_T`. By default, generated code represents the Simulink data type `single` by using the identifier `real32_T`.

The code also provides a macro guard of the form `RTW_HEADER_filename_h_`. When you export a data type definition to integrate generated code with code from other sources, you can use macro guards of this form to prevent identifier clashes.

```
file = fullfile('rtwdemo_basicsc_ert_rtw','myHdrFile.h');
rtwdemodbtype(file,'#ifndef RTW_HEADER_myHdrFile_h_',...
    '/* RTW_HEADER_myHdrFile_h_ */',1,1)
```

```
#ifndef RTW_HEADER_myHdrFile_h_
#define RTW_HEADER_myHdrFile_h_
#include "rtwtypes.h"

typedef real32_T mySingleAlias;
typedef creal32_T cmySingleAlias;
```

View the file `rtwdemo_basicsc.h`. The code uses the data type alias `mySingleAlias` to define the structure field `input1`, which corresponds to the Inport block labeled `In1`.

```
file = fullfile('rtwdemo_basicsc_ert_rtw','rtwdemo_basicsc.h');
rtwdemodbtype(file,'/* External inputs (root inport signals with auto storage) */',...
    '} ExtU_rtwdemo_basicsc_T;',1,1)
```

```
/* External inputs (root inport signals with auto storage) */
typedef struct {
  mySingleAlias input1;                /* '<Root>/In1' */
  real32_T input2;                     /* '<Root>/In2' */
  real32_T input3;                     /* '<Root>/In3' */
  real32_T input4;                     /* '<Root>/In4' */
```

```
} ExtU_rtwdemo_basicsc_T;
```

Import Type Definition

When you integrate code generated from a model with code from other sources, to avoid redundant `typedef` statements, you can import a data type definition to the model code. This example shows how to import your own definition of a data type from a header file that you create.

Use a text editor to create a header file to import. Name the file `myImportedHdrFile.h`. Place it in your working folder. Copy the following code into the file.

```
#ifndef HEADER_myImportedHdrFile_h_
#define HEADER_myImportedHdrFile_h_

typedef float myTypeAlias;

#endif
```

The code uses the identifier `myTypeAlias` to create an alias for the data type `float`. The code also uses a macro guard of the form `HEADER_filename_h`. When you import a data type definition to integrate generated code with code from other sources, you can use macro guards of this form to prevent identifier clashes.

At the command prompt, create a `Simulink.AliasType` object named `myTypeAlias` that creates an alias for the built-in type `single`. The Simulink data type `single` corresponds to the data type `float` in generated code.

```
myTypeAlias = Simulink.AliasType('single')
```

```
myTypeAlias =

  AliasType with properties:

    Description: ''
      DataScope: 'Auto'
     HeaderFile: ''
       BaseType: 'single'
```

Configure the object so that generated code imports the type definition from the header file `myImportedHdrFile.h`.

```
myTypeAlias.DataScope = 'Imported';
myTypeAlias.HeaderFile = 'myImportedHdrFile.h';
```

Open the model `rtwdemo_basicsc`.

```
open_system('rtwdemo_basicsc')
```

Inspect the parameters of the Inport block labeled `In1`. On the **Signal Attributes** tab, **Data type** is set to `single`.

Set **Data type** to the alias `myTypeAlias`.

```
set_param('rtwdemo_basicsc/In1','OutDataTypeStr','myTypeAlias')
```

In the model, set **Configuration Parameters > Code Generation > System target file** to `ert.tlc`. With this setting, the code generator honors data type aliases such as `myTypeAlias`.

```
set_param('rtwdemo_basicsc','SystemTargetFile','ert.tlc')
```

Generate code from the model.

```
rtwbuild('rtwdemo_basicsc')
```

```
### Starting build procedure for model: rtwdemo_basicsc
### Successful completion of build procedure for model: rtwdemo_basicsc
```

In the code generation report, view the file `rtwdemo_basicsc_types.h`. The code creates a `#include` directive for your header file `myImportedHdrFile.h`.

```
file = fullfile('rtwdemo_basicsc_ert_rtw','rtwdemo_basicsc_types.h');
rtwdemodbtype(file,'#include "myImportedHdrFile.h',...
    '/* Forward declaration for rtModel */',1,0)
```

```
#include "myImportedHdrFile.h"
```

View the file `rtwdemo_basicsc.h`. The code uses the data type alias `myTypeAlias` to define the structure field `input1`, which corresponds to the Inport block labeled `In1`.

```
file = fullfile('rtwdemo_basicsc_ert_rtw','rtwdemo_basicsc.h');
rtwdemodbtype(file,'/* External inputs (root inport signals with auto storage) */',...
    '} ExtU_rtwdemo_basicsc_T;',1,1)
```

```
/* External inputs (root inport signals with auto storage) */
typedef struct {
  myTypeAlias input1;                  /* '<Root>/In1' */
  real32_T input2;                     /* '<Root>/In2' */
  real32_T input3;                     /* '<Root>/In3' */
  real32_T input4;                     /* '<Root>/In4' */
} ExtU_rtwdemo_basicsc_T;
```

Display Base Data Types and Aliases on Model Diagram

When you display signal data types on the model diagram by selecting **Display > Signals and Ports > Port Data Types**, by default, the diagram displays aliases instead of base data types (such as `int16`). To display the base types, choose an option for **Display > Signals and Ports > Port Data Type Display Format**. For more information, see "Port Data Types" (Simulink).

See Also

`Simulink.AliasType` | `Simulink.importExternalCTypes` | `Simulink.NumericType`

Related Examples

- "Conform to Coding Standards by Replacing and Renaming Data Types" on page 21-22
- "Create and Apply User-Defined Data Types" on page 21-9
- "Data Type Replacement" on page 21-36
- "Use `single` Data Type as Default for Underspecified Types" on page 19-43
- "Create a Named Fixed-Point Data Type in the Generated Code" on page 21-18
- "What Are User-Defined Data Types?" on page 21-2
- "Group Signals into Structures in the Generated Code Using Buses" on page 19-139
- "Data Objects" (Simulink)

Create a Named Fixed-Point Data Type in the Generated Code

This example shows how to create and name a fixed-point data type in generated code. You can use the name of the type to specify parameter and signal data types throughout a model and in generated code.

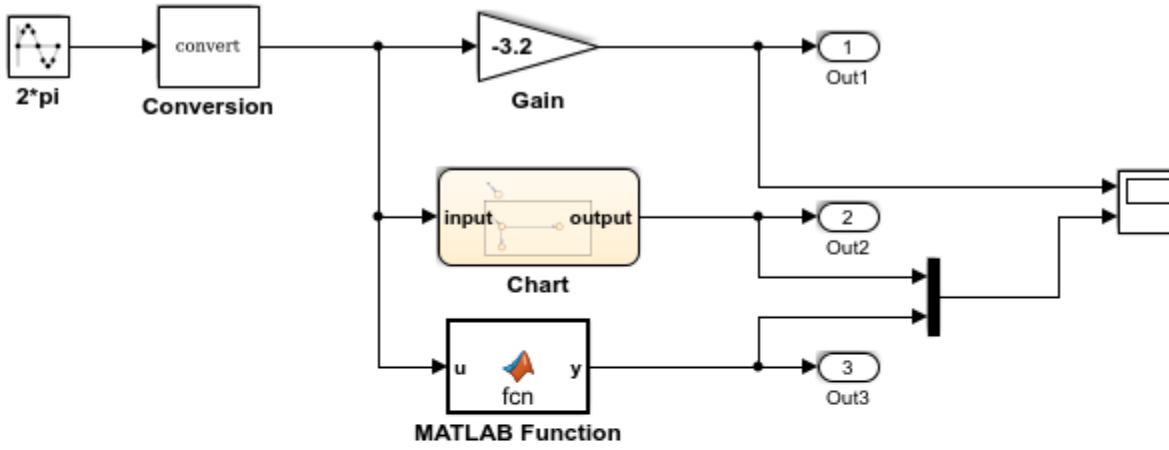
The example model `rtwdemo_fixpt1` uses fixed-point data types. So that you can more easily see the fixed-point data type in the generated code, in this example, you create a `Simulink.Parameter` object that appears in the code as a global variable.

Create a `Simulink.AliasType` object that defines a fixed-point data type. Name the object `myFixType`. The generated code uses the name of the object as a data type.

```
myFixType = Simulink.AliasType('fixdt(1,16,4)');
```

Open the model `rtwdemo_fixpt1`.

```
open_system('rtwdemo_fixpt1')
```



Inspect the parameters of the Gain block. For example, open the block dialog box.

Set the value of the **Gain** parameter to `myKParam`.

Click the action button next to the parameter value. Select **Create Variable**.

In the **Create New Data** dialog box, set `Value` to `Simulink.Parameter(8)`. Click **Create**. A `Simulink.Parameter` object named `myKParam` appears in the base workspace. The object stores the real-world value `8`, which the Gain block uses for the value of the **Gain** parameter.

In the **Simulink.Parameter** property dialog box, set **Storage class** to `ExportedGlobal`. Click **OK**. With this setting, `myKParam` appears in the generated code as a separate global variable.

In the block dialog box, on the **Signal Attributes** tab, set **Output data type** to `myFixType`.

On the **Parameter Attributes** tab, set **Parameter data type** to `myFixType`.

Alternatively, you can use these commands at the command prompt to configure the block and create the object:

```
set_param('rtwdemo_fixpt1/Gain','Gain','myKParam','OutDataTypeStr','myFixType',...
    'ParamDataTypeStr','myFixType')
myKParam = Simulink.Parameter(8);
myKParam.StorageClass = 'ExportedGlobal';
```

Inspect the parameters of the Data Type Conversion block labeled `Conversion`. Set **Output data type** to `myFixType`.

```
set_param('rtwdemo_fixpt1/Conversion','OutDataTypeStr','myFixType')
```

In the model, set **Configuration Parameters > Code Generation > System target file** to `ert.tlc`. With this setting, the code generator honors data type aliases such as `myFixType`.

```
set_param('rtwdemo_fixpt1','SystemTargetFile','ert.tlc')
```

Select the configuration parameter **Generate code only**.

```
set_param('rtwdemo_fixpt1','GenCodeOnly','on')
```

Generate code from the model.

```
rtwbuild('rtwdemo_fixpt1')
```

```
### Starting build procedure for model: rtwdemo_fixpt1
### Successful completion of code generation for model: rtwdemo_fixpt1
```

In the code generation report, view the file `rtwdemo_fixpt1_types.h`. The code defines the type `myFixType` based on an integer type of the specified word length (16).

```
file = fullfile('rtwdemo_fixpt1_ert_rtw','rtwdemo_fixpt1_types.h');
rtwdemodbtype(file,'#ifndef DEFINED_TYPEDEF_FOR_myFixType_',...
    '/* Forward declaration for rtModel */',1,0)
```

```
#ifndef DEFINED_TYPEDEF_FOR_myFixType_
#define DEFINED_TYPEDEF_FOR_myFixType_

typedef int16_T myFixType;
typedef cint16_T cmyFixType;

#endif
```

View the file `rtwdemo_fixpt1.c`. The code uses the type `myFixType`, which is an alias of the integer type `int16`, to define the variable `myKParam`.

```
file = fullfile('rtwdemo_fixpt1_ert_rtw','rtwdemo_fixpt1.c');
rtwdemodbtype(file,'myFixType myKParam = 128;','myFixType myKParam = 128;',1,1)
```

```
myFixType myKParam = 128;              /* Variable: myKParam
```

The stored integer value `128` of `myKParam` is not the same as the real-world value `8` because of the scaling that the fixed-point data type `myFixType` specifies. For more information, see "Scaling" (Fixed-Point Designer) in the Fixed-Point Designer documentation.

The line of code that represents the Gain block applies a right bit shift corresponding to the fraction length specified by `myFixType`.

```
rtwdemodbtype(file,...
    'rtwdemo_fixpt1_Y.Out1 = (myFixType)(myKParam * rtb_Conversion >> 4);',...
    'rtwdemo_fixpt1_Y.Out1 = (myFixType)(myKParam * rtb_Conversion >> 4);',1,1)
```

```
  rtwdemo_fixpt1_Y.Out1 = (myFixType)(myKParam * rtb_Conversion >> 4);
```

See Also

`fixdt` | `Simulink.NumericType`

Related Examples

- "Air-Fuel Ratio Control System with Fixed-Point Data" (Simulink Coder)
- "Create and Apply User-Defined Data Types" on page 21-9
- "What Are User-Defined Data Types?" on page 21-2
- "Data Objects" (Simulink)

Conform to Coding Standards by Replacing and Renaming Data Types

By default, the generated code uses Simulink Coder data type aliases such as `real_T` and `int32_T`. The code uses these aliases to define global and local variables. If your coding standards require that you use other data type aliases, including aliases that your existing code defines, you can:

- Configure data type replacement for the entire model.
- Configure individual data items (such as signals, parameters, and states) to use specific data type aliases.

For basic information about controlling data types in a model, see "Control Signal Data Types" (Simulink).

Inspect Custom C Code

Save this custom C code into a file named `my_types.h` in your current folder. This file represents a header file in your existing code that defines custom data type aliases by using `typedef` statements.

```
#include <stdbool.h>

typedef double my_dblPrecision;
typedef short my_int16;
typedef bool my_bool;
```

Explore Example Model and Default Generated Code

1. Open the example model ex_data_type_replacement.

   ```
   open_system(fullfile(docroot,'toolbox','ecoder','examples',...
   'ex_data_type_replacement'))
   ```

2. Right-click one of the named signal lines, such as `temp`, and select **Properties**.
3. In the Signal Properties dialog box, select the **Code Generation** tab.

   For the named signals, **Storage class** is set to `ExportedGlobal`. With this setting, the signal lines appear in the generated code as separate global variables.
4. Update the block diagram.

The signals in the model use a mix of the data types `int16`, `double`, and `boolean`.

5 Generate code from the model.

6 In the code generation report, inspect the shared utility file `rtwtypes.h`. The code uses `typedef` statements to rename the primitive C data types by using standard Simulink Coder aliases. For example, the code renames the primitive type `double` by using the alias `real_T`.

```
typedef double real_T;
```

7 Inspect the file `ex_data_type_replacement.c`. The code uses the Simulink Coder data type aliases to declare and define variables. For example, the code uses the data types `real_T`, `int16_T`, and `boolean_T` to define the global variables `flowIn`, `temp`, and `intlk`.

```
real_T flowIn;                          /* '<Root>/In3' */
int16_T temp;                           /* '<Root>/Add2' */
boolean_T intlk;                        /* '<S1>/Compare' */
```

The model `step` function defines local variables by using the same data type aliases.

```
real_T rtb_Add;
real_T rtb_FilterCoefficient;
```

Reuse Custom Data Type Definitions

1 At the command prompt, create a `Simulink.AliasType` object for each data type alias that your custom code defines.

```
Simulink.importExternalCTypes('my_types.h');
```

In the base workspace, the `Simulink.importExternalCTypes` function creates the objects `my_dblPrecision`, `my_int16`, and `my_bool`.

For each object, the function sets the `DataScope` property to `'Imported'` and the `HeaderFile` property to `'my_types.h'`. With these settings, the code generator does not create a `typedef` statement for each object, but instead the generated code reuses the statements from `my_types.h` by including (`#include`) the file.

2 In the model, in the Configuration Parameters dialog box, on the **Code Generation > Data Type Replacement** pane, select **Replace data type names in the generated code**.

3 Specify the options in the **Replacement Name** column according to the table.

Simulink Name	Replacement Name
`double`	`my_dblPrecision`
`int16`	`my_int16`
`boolean`	`my_bool`

4 Generate code from the model.

5 In the code generation report, inspect the file `rtwtypes.h`. Now, the code uses an `#include` directive to import the contents of the custom header file `my_types.h`, which contains the custom data type aliases.

```
#include "my_types.h"                  /* User defined replacement datatype for int
```

6 Inspect the file `ex_data_type_replacement.c`. The code uses the custom data type aliases `my_dblPrecision`, `my_int16`, and `my_bool` to define the global variables such as `flowIn`, `temp`, and `intlk`.

```
my_dblPrecision flowIn;                /* '<Root>/In3' */
my_int16 temp;                         /* '<Root>/Add2' */
my_bool intlk;                         /* '<S1>/Compare' */
```

The model `step` function defines local variables by using the custom data type aliases.

```
my_dblPrecision rtb_Add;
my_dblPrecision rtb_FilterCoefficient;
```

Create Meaningful Data Type Aliases for Individual Data Items

Suppose your coding standards require that important data items use a data type whose name indicates the real-world significance. You can create more `Simulink.AliasType` objects to represent these custom data type aliases. Use the objects to set the data types of data items in the model.

1 In the model, set these block parameters.

Block	Parameter	Parameter Value
In3	**Data type**	`flow_T`
Flow Setpoint	**Output data type**	`flow_T`

Block	Parameter	Parameter Value
Add2	**Output data type**	`diffTemp_T`
Flow Controller	**Sum output**	`ctrl_T`

With these settings, some of the named signals, such as `temp` and `flowIn`, use data types that evoke real-world quantities, such as liquid flow rate and temperature.

2. At the command prompt, create `Simulink.AliasType` objects to represent these new custom data type aliases.

```
flow_T = Simulink.AliasType('double');
diffTemp_T = Simulink.AliasType('int16');
ctrl_T = Simulink.AliasType('double');
```

In the model, the signals `flowIn` and `flowSetPt` use the primitive data type `double`, so the data type alias `flow_T` maps to `double`.

3. Update the block diagram.

Due to data type inheritance, other signals also use the custom data type aliases. For example, in the Add block dialog box, on the **Signal Attributes** tab, the **Output data type** parameter is set to the default value, `Inherit: Inherit via internal rule`. The internal rule chooses the same data type that the block inputs use, `flow_T`.

4. Generate code from the model.

5. The file `ex_data_type_replacement_types.h` defines the new data types `flow_T`, `diffTemp_T`, and `ctrl_T` as aliases of `my_dblPrecision` and `my_int16`.

```
typedef my_dblPrecision flow_T;
typedef my_int16 diffTemp_T;
typedef my_dblPrecision ctrl_T;
```

6. In the file `ex_data_type_replacement.c`, the code defines global variables by using the new type names.

```
flow_T flowIn;                          /* '<Root>/In3' */
flow_T flowSetPt;                       /* '<Root>/Flow Setpoint' */
ctrl_T flowCtrl;                        /* '<Root>/Interlock' */
diffTemp_T temp;                            /* '<Root>/Add2' */
```

7. For blocks that do not use the new data types, the corresponding generated code continues to use the replacement types that you specified earlier. For example, in the

file `ex_data_type_replacement.h`, the blocks In1 and In2 appear as structure fields that use the replacement type `my_int16`.

```
/* External inputs (root inport signals with auto storage) */
typedef struct {
  my_int16 In1;                        /* '<Root>/In1' */
  my_int16 In2;                        /* '<Root>/In2' */
} ExtU_ex_data_type_replacement_T;
```

Create Single Point of Definition for Primitive Types

The custom data type aliases `flow_T` and `ctrl_T` map to the primitive data type `double`. If you want to change this underlying data type from `double` to `single` (`float`), you must remember to modify the `BaseType` property of both `Simulink.AliasType` objects.

To more easily make this change, you can create a `Simulink.NumericType` object and configure both `Simulink.AliasType` objects to refer to it. Then, you need to modify only the `Simulink.NumericType` object. A `Simulink.NumericType` object enables you to share a data type without creating a data type alias.

1. At the command prompt, create a `Simulink.NumericType` object to represent the primitive data type `single`.

   ```
   sharedType = Simulink.NumericType;
   sharedType.DataTypeMode = 'Single';
   ```

2. Configure the `Simulink.AliasType` objects `flow_T` and `ctrl_T` to acquire an underlying data type from this new object.

   ```
   flow_T.BaseType = 'sharedType';
   ctrl_T.BaseType = 'sharedType';
   ```

3. In the model, select **Display > Signals and Ports > Port Data Type Display Format > Base and Alias Types** (see "Port Data Types" (Simulink)). Update the block diagram.

   The data type indicators in the model show that the aliases `flow_T` and `ctrl_T` map to the primitive type `single`. To change this underlying primitive type, you can modify the `DataTypeMode` property of the `Simulink.NumericType` object, `sharedType`.

By default, the `Simulink.NumericType` object does not cause another `typedef` statement to appear in the generated code.

If you generate code from the model while the `Simulink.NumericType` object represents the data type `single`, the generated code maps `flow_T` and `ctrl_T` to the default Simulink Coder data type alias `real32_T`, which maps to the C data type `float`. You can replace `real32_T` in the same way that you replaced `real_T`, `int16_T`, and `boolean_T` (**Configuration Parameters** > **Code Generation** > **Data Type Replacement**).

Permanently Store Data Type Objects

The `Simulink.NumericType` and `Simulink.AliasType` objects in the base workspace do not persist if you end your current MATLAB session. To permanently store these objects, consider migrating your model to a data dictionary. See "Migrate Models to Use Simulink Data Dictionary" (Simulink).

Create and Maintain Objects Corresponding to Multiple C `typedef` Statements

To create `Simulink.AliasType` objects for a large number of `typedef` statements in your external C code, consider using the `Simulink.importExternalCTypes` function.

Related Examples

- "Control Code Style" on page 36-36
- "Unit Specification in Simulink Models" (Simulink)
- "Design Data Interface by Configuring Inport and Outport Blocks" on page 19-134
- "Choose an External Code Integration Workflow" on page 39-4
- "Data Type Replacement" on page 21-36
- "Create Data Type Alias in the Generated Code" on page 21-12
- "Control File Placement of User-Defined Types" on page 21-6

Exchange Structured and Enumerated Data Between Generated and External Code

This example shows how to generate code that exchanges data with some external, handwritten code. You construct and configure a model to match data types with the external code and to avoid duplicating type definitions and memory allocation (definition of global variables). You then compile the generated code together with the external code into a single application.

Inspect External Code

Create the file `ex_cc_algorithm.c` in your current folder.

```
#include "ex_cc_algorithm.h"

inSigs_T inSigs;

float_32 my_alg(void)
{
    if (inSigs.err == TMP_HI) {
        return 27.5;
    }
    else if (inSigs.err == TMP_LO) {
        return inSigs.sig1 * calPrms.cal3;
    }
    else {
        return inSigs.sig2 * calPrms.cal3;
    }
}
```

The C code defines a function, `my_alg`, that uses global structure variables such as `inSigs` and `calPrms`. The code also allocates memory for `inSigs`.

Create the file `ex_cc_algorithm.h` in your current folder.

```
#ifndef ex_cc_algorithm_h
#define ex_cc_algorithm_h

typedef float float_32;
```

```
typedef enum {
    TMP_HI = 0,
    TMP_LO,
    NORM,
} err_T;

typedef struct inSigs_tag {
    err_T err;
    float_32 sig1;
    float_32 sig2;
} inSigs_T;

typedef struct calPrms_tag {
    float_32 cal1;
    float_32 cal2;
    float_32 cal3;
} calPrms_T;

extern calPrms_T calPrms;
extern inSigs_T inSigs;

float_32 my_alg(void);

#endif
```

The file defines `float_32` as an alias of the C data type `float`. The file also defines an enumerated data type and two structure types.

The function `my_alg` is designed to calculate a return value by using the fields of `inSigs` and `calPrms`, which are global structure variables of the types `inSigs_T` and `calPrms_T`. The function requires another algorithm to supply the signal data that `inSigs` stores.

This code allocates memory for `inSigs`, but not for `calPrms`. You must create a model whose generated code:

- Defines and initializes `calPrms`.
- Calculates values for the fields of `inSigs`.
- Reuses the type definitions (such as `err_T` and `float_32`) that the external code defines.

Create Simulink Model

So that you can create enumerated and structured data in the Simulink model, first create Simulink representations of the data types that the external code defines. Store the Simulink types in a new data dictionary named `ex_cc_integ.sldd`.

```
Simulink.importExternalCTypes('ex_cc_algorithm.h',...
    'DataDictionary','ex_cc_integ.sldd');
```

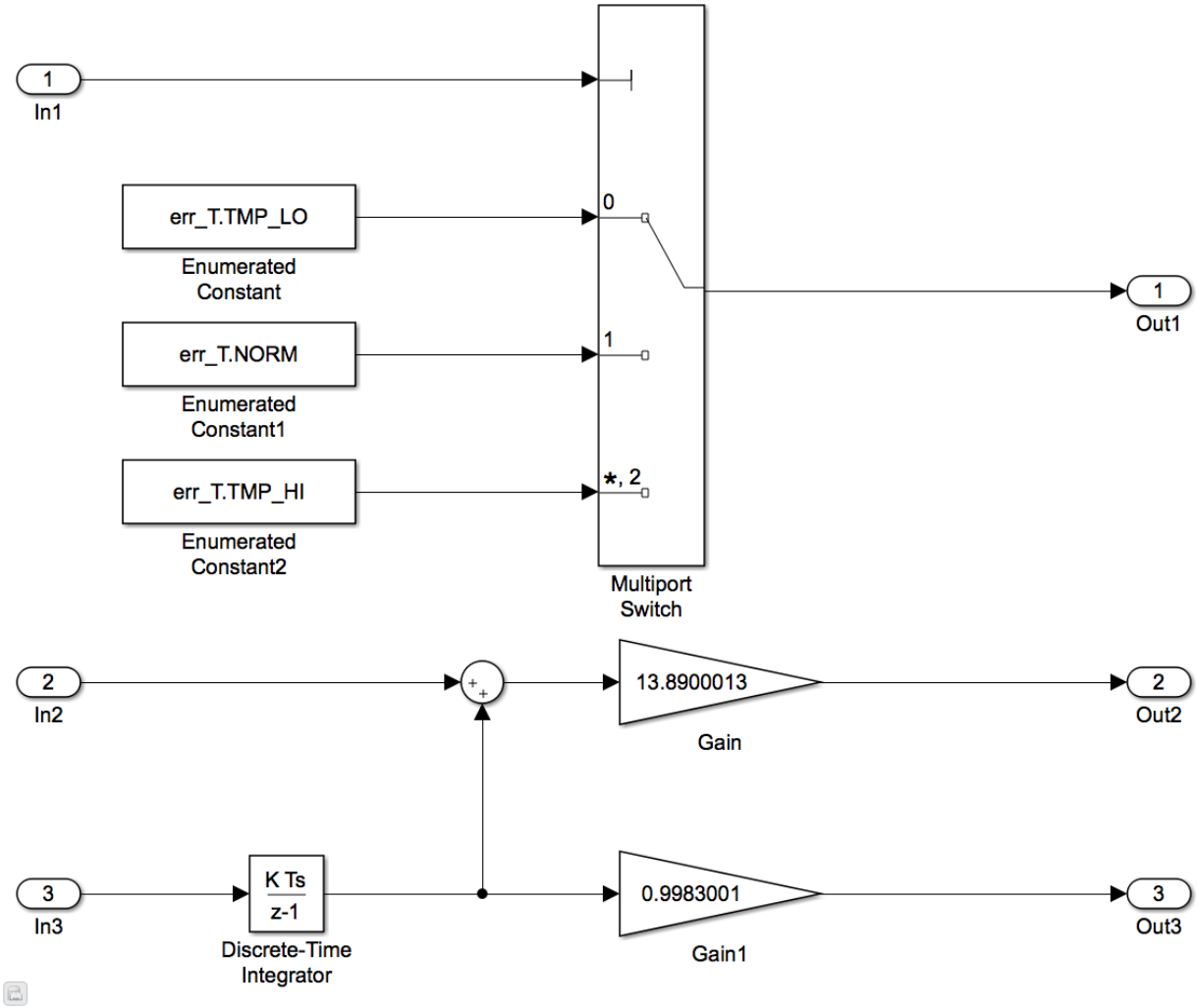
The data dictionary appears in your current folder.

In your current folder, double-click the file to inspect the dictionary contents in the Model Explorer. The `Simulink.importExternalCTypes` function creates `Simulink.Bus`, `Simulink.AliasType`, and `Simulink.data.dictionary.EnumTypeDefinition` objects that correspond to the custom C data types from `ex_cc_algorithm.h`.

Create a new, empty model named `ex_struct_enum_integ`.

Link the model to the data dictionary. In the model, select **File** > **Model Properties** > **Link to Data Dictionary**.

Add algorithmic blocks that calculate the fields of `inSigs`.



Now that you have the basic algorithmic model, you must:

- Organize the output signals into a structure variable named `inSigs`.
- Create the structure variable `calPrms`.
- Include `ex_cc_algorithm.c` in the build process that compiles the code after code generation.

Configure Generated Code to Write Outputs to Existing Structure Variable

Add a Bus Creator block near the existing Outport blocks.

In the Bus Creator block, set these parameters:

- **Number of inputs** to `3`
- **Output data type** to `Bus: inSigs_T`
- **Output as nonvirtual bus** to selected

Delete the three existing Outport blocks (but not the signals that enter the blocks).

Connect the three remaining signal lines to the inputs of the Bus Creator block.

Add an Outport block after the Bus Creator block. Connect the output of the Bus Creator to the Outport.

In the Outport block, set the **Data type** parameter to `Bus: inSigs_T`.

In the model, select **View** > **Model Data**.

On the **Inports/Outports** tab, for the Inport blocks labeled `In2` and `In3`, change **Data Type** from `Inherit: auto` to `float_32`.

Change the **Change View** drop-down list from `Design` to `Code`.

For the Outport block, set **Signal Name** to `inSigs`.

Set **Storage Class** to `ImportFromFile`.

Set **Header File** to `ex_cc_algorithm.h`.

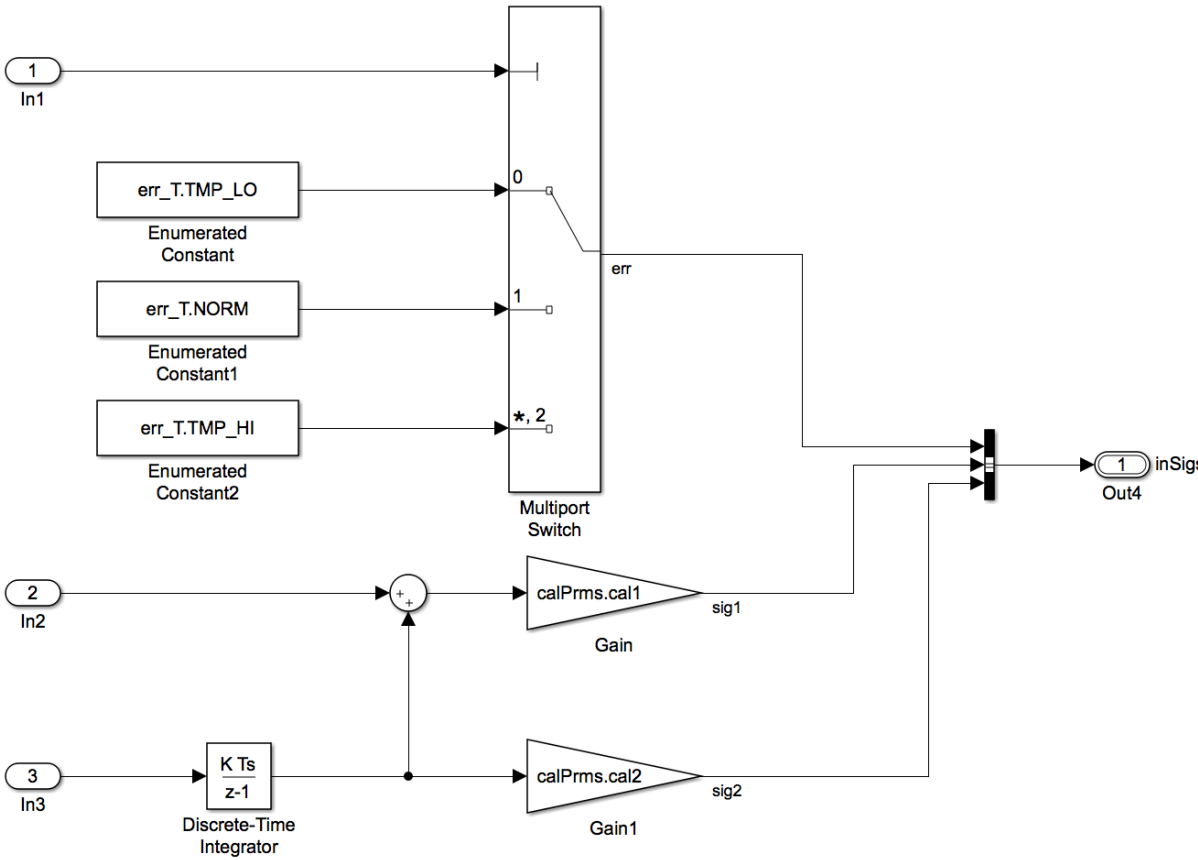
Inspect the **Signals** tab.

In the model, select the output signal of the Multiport Switch block.

In the Model Data Editor, for the selected signal, set **Name** to `err`.

Set the name of the output signal of the Gain block to `sig1`. Set the name of the output signal of the Gain1 block to `sig2`.

When you finish, the model stores output signal data (such as the signals `err` and `sig1`) in the fields of a structure variable named `inSigs`.



Because you set **Storage Class** to `ImportFromFile`, the generated code does not allocate memory for `inSigs`.

Configure Model to Generate Parameter Data

In the Model Explorer **Model Hierarchy** pane, under the dictionary node **ex_cc_integ**, select the **Design Data** node.

In the **Contents** pane, select the `Simulink.Bus` object `calPrms_T`.

In the Dialog pane (the right pane), click **Launch Bus Editor**.

In the Bus Editor, in the left pane, select `calPrms_T`.

On the toolbar, click the **Create/Edit a Simulink.Parameter Object from a Bus Object** button.

In the MATLAB Editor, copy the generated MATLAB code and run the code at the command prompt. The code creates a `Simulink.Parameter` object in the base workspace.

At the command prompt, set the field values in the parameter object. For the fields `cal1` and `cal2`, use the same values as the Gain blocks in the model. For `cal3`, use a nonzero number such as `15.2299995`.

```
calPrms_T_Param.Value.cal1 = 13.8900013;
calPrms_T_Param.Value.cal2 = 0.9983001;
calPrms_T_Param.Value.cal3 = 15.2299995;
```

In the Model Explorer **Model Hierarchy** pane, select **Base Workspace**.

Move the parameter object from the base workspace to the Design Data section of the data dictionary.

With the data dictionary selected, in the **Contents** pane, rename the parameter object as `calPrms`.

In the right pane, set **Storage class** to `ExportedGlobal`.

Save the changes that you made to the dictionary.

In the Model Data Editor, select the **Parameters** tab.

For the Gain block, replace the value `13.8900013` with `calPrms.cal1`.

In the other Gain block, use `calPrms.cal2`.

Generate, Compile, and Inspect Code

Configure the model to include `ex_cc_algorithm.c` in the build process. Set **Configuration Parameters** > **Code Generation** > **Custom Code** > **Additional build information** > **Source files** to `ex_cc_algorithm.c`.

Generate code from the model.

Inspect the generated file `ex_struct_enum_integ.c`. The file defines and initializes `calPrms`.

```
/* Exported block parameters */
calPrms_T calPrms = {
  13.8900013F,
  0.998300076F,
  15.23F
} ;                                    /* Variable: calPrms
```

The generated algorithm in the model `step` function defines a local variable for buffering the value of the signal `err`.

```
err_T rtb_err;
```

The algorithm then calculates and stores data in the fields of `inSig`.

```
inSigs.err = rtb_err;
inSigs.sig1 = (rtU.In2 + rtDW.DiscreteTimeIntegrator_DSTATE) * calPrms.cal1;
inSigs.sig2 = calPrms.cal2 * rtDW.DiscreteTimeIntegrator_DSTATE;
```

Replace Data Type Names Throughout Model

To generate code that uses `float_32` instead of the default, `real32_T`, instead of manually specifying the data types of block output signals and bus elements, you can use data type replacement (**Configuration Parameters** > **Code Generation** > **Data Type Replacement**). For more information, see "Conform to Coding Standards by Replacing and Renaming Data Types" on page 21-22.

See Also

`Simulink.importExternalCTypes`

Related Examples

- "Exchange Data Between External C/C++ Code and Simulink Model or Generated Code" (Simulink Coder)
- "What Are User-Defined Data Types?" on page 21-2
- "Use Enumerated Data in Generated Code" on page 19-22
- "Group Signals into Structures in the Generated Code Using Buses" on page 19-139

Data Type Replacement

In this section...

When you generate code for a model, you can replace the default Simulink Coder data type names, such as `real_T` and `boolean_T`, with your own custom names. The model code creates `typedef` statements to define your replacement names. It uses your replacement names instead of the default type names to, for example, define variables and functions.

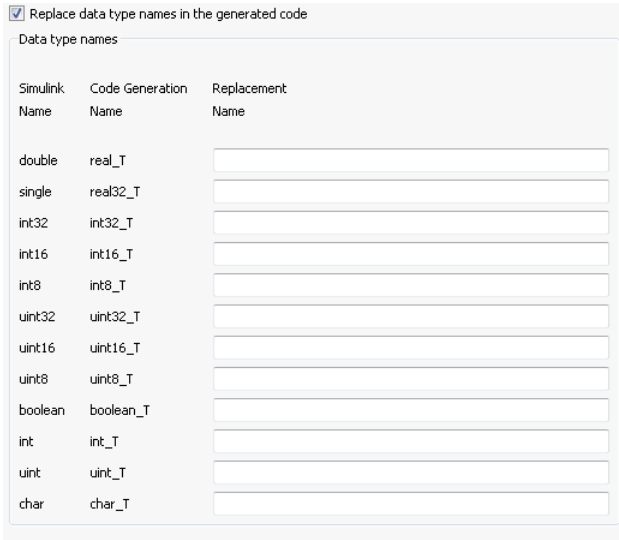
You can specify many-to-one data type replacement to replace multiple built-in data types with one name in the generated code. For example, you can replace the built-in data types `uint8` and `boolean` with a single data type name that you specify.

In generated code, data type replacement uses the replacements that you specify instead of the default Simulink Coder data type names. If you want to create custom data type names for individual block parameters and signals in generated code and in a block diagram, see “Create Data Type Alias in the Generated Code” on page 21-12.

Replace Built-In Data Types

To configure replacement data type names:

1. In the Configuration Parameters dialog box, select **Code Generation > Data Type Replacement** and **Replace data type names in the generated code**. The **Data type names** table lists each Simulink built-in data type name with the corresponding code generation name.



2 Specify the **Replacement Name** column with values that replace the default names in the **Code Generation Name** column. Specify one of these options:

- The name of a `Simulink.AliasType` object that is in the base workspace or a data dictionary. When you use a `Simulink.AliasType` object, you can replace a data type name with the name of the object.

  Set the `BaseType` property of the object to the corresponding **Simulink Name** data type. Set the `DataScope` property of the object to `Auto` (default) or `Imported`. If you want to use your own header file to define replacement names, set the `HeaderFile` property of the object to the header file name and set `DataScope` to `Imported`.
- The data type name from the **Simulink Name** column. This name replaces the data type name in the generated code. Using the **Simulink Name**, you can replace data types except `real_T` and `real32_T`. To specify replacement names for `boolean_T`, `int_T`, `uint_T`, and `char_T`, see the following table.
- The name of a `Simulink.NumericType` object that is in the base workspace or a data dictionary. When you use a `Simulink.NumericType` object, you can define replacement names for `real_T`, `real32_T`, and `boolean_T`.

Set the `DataTypeMode` property of the object to the corresponding data type name from the **Simulink Name** column.

Specify the Replacement Name for a Data Type

To replace the Code Generation Name	Specify a `Simulink.AliasType` object with `BaseType`	Specify the corresponding Simulink Name	Specify a `Simulink.NumericType` object with `DataTypeMode`
`real_T`	`double`	–	`Double`
`real32_T`	`single`	–	`Single`
`int32_T`	`int32`	`int32`	–
`int16_T`	`int16`	`int16`	–
`int8_T`	`int8`	`int8`	–
`uint32_T`	`uint32`	`uint32`	–
`uint16_T`	`uint16`	`uint16`	–
`uint8_T`	`uint8`	`uint8`	–
`boolean_T`	`uint8` or `int8` or `intn*`	`uint8` or `int8` or `intn*`	`Boolean`
`int_T`	`intn*`	`intn*`	–
`uint_T`	`uintn*`	`uintn*`	–
`char_T`	`intn*`	`intn*`	–

* Replace *n* with the number of bits displayed in the Configuration Parameters dialog box **Hardware Implementation** pane in either **Number of bits: int** or **Number of bits: char**, depending on the data type that you want to replace.

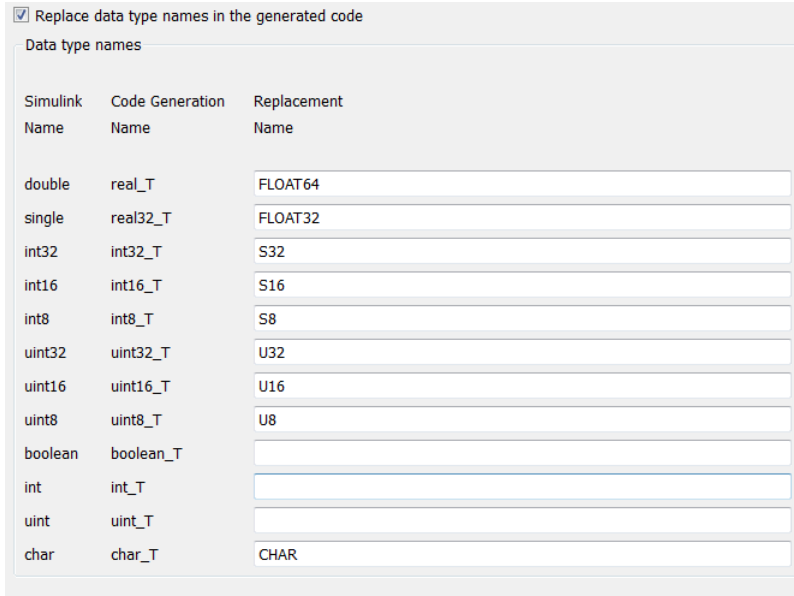
Note: The `boolean_T` `BaseType` must promote to a signed `int`.

Suppose that in the base workspace you define these replacement data types as `Simulink.AliasType` objects.

Replacement Name	Description
`FLOAT64`	64-bit floating point

Replacement Name	Description
FLOAT32	32-bit floating point
S32	32-bit integer
S16	16-bit integer
S8	8-bit integer
U32	Unsigned 32-bit integer
U16	Unsigned 16-bit integer
U8	Unsigned 8-bit integer
CHAR	Character data

You can specify data type replacements with a one-to-one replacement mapping as shown.

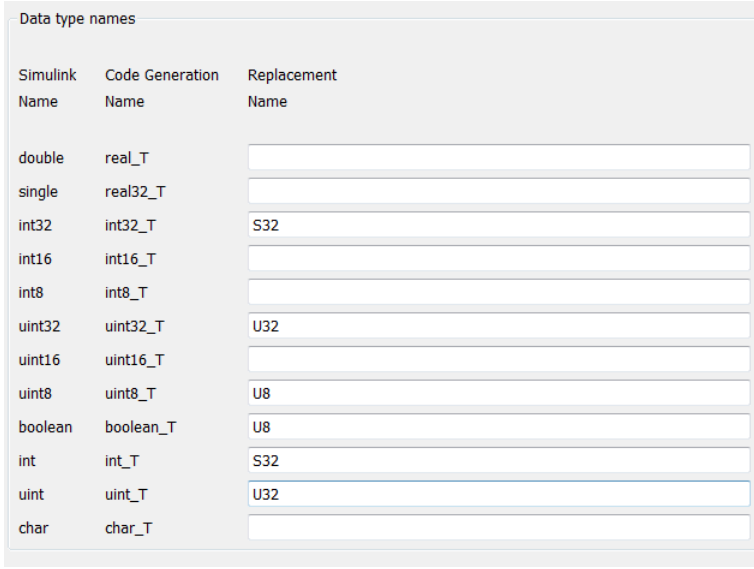


You can also apply a many-to-one data type replacement mapping. For example, you can replace these data types:

- `int32` and `int` with the name `S32`.

- `uint32` and `uint` with the name `U32`.
- `uint8` and `boolean` with the name `U8`.

Note: Many-to-one data type replacement does not support the `char` (`char_T`) built-in data type. Use `char` only in one-to-one data type replacements.



Programmatically Replace Built-In Data Types

To programmatically replace the built-in data type names for your model, adjust the `ReplacementTypes` model parameter, which is a structure. This example code shows how to modify the `ReplacementTypes` parameter to replace the built-in data type names `int8`, `uint8`, and `boolean` with the custom data type names `my_T_S8`, `my_T_U8`, and `my_T_BOOL`.

```
model = bdroot;
cs = getActiveConfigSet(model);
set_param(cs,'EnableUserReplacementTypes','on');

struc = get_param(cs,'ReplacementTypes');
struc.int8 = 'my_T_S8';
```

```
struc.uint8 = 'my_T_U8';
struc.boolean = 'my_T_BOOL';

set_param(cs,'ReplacementTypes',struc);
```

Data Type Replacement Limitations

When you select the model configuration parameter **Replace data type names in the generated code** on the **Code Generation > Data Type Replacement** pane of the Configuration Parameters dialog box, these limitations apply.

- Data type replacement does not support multiple levels of mapping. Each replacement data type name maps directly to one or more built-in data types.
- Data type replacement is not supported for simulation target code generation for referenced models.
- If you select the **Classic call interface** configuration parameter for your model, data type replacement is not supported.
- Code generation performs data type replacements while generating `.c`, `.cpp`, and `.h` files. Code generation places these files in build folders (including top and referenced model build folders) and in the `_sharedutils` folder. *Exceptions* are as follows:
  `rtwtypes.h`
  `multiword_types.h`
  `model_reference_types.h`
  `builtin_typeid_types.h`
  `model_sf.c` or `.cpp` (ERT S-function wrapper)
  `model_dt.h` (C header file supporting external mode)
  `model_capi.c` or `.cpp`
  `model_capi.h`
- Data type replacement is not supported for complex data types.
- Many-to-one data type replacement is not supported for the `char` data type. Attempting to use `char` as part of a many-to-one mapping to a custom data type represents a violation of the MISRA C specification. For example, if you map `char` (`char_T`) and either `int8` (`int8_T`) or `uint8` (`uint8_T`) to the same replacement type, the result is a MISRA C violation. If you try to generate C++ code, the code generator makes invalid implicit type casts, resulting in compile-time errors. Use `char` only in one-to-one data type replacements.
- For ERT S-functions, replace the `boolean` data type with only an 8-bit integer, `int8`, or `uint8`.

- Set the `DataScope` property of a `Simulink.AliasType` object to `Auto` (default) or `Imported`.

See Also

`Simulink.AliasType` | `Simulink.NumericType`

Related Examples

- “Conform to Coding Standards by Replacing and Renaming Data Types” on page 21-22
- “Replace `boolean` with Specific Integer Data Type” on page 56-14
- “Create Data Type Alias in the Generated Code” on page 21-12
- “What Are User-Defined Data Types?” on page 21-2

Specify Boolean and Data Type Limit Identifiers

You can use command-line parameters to replace the default Boolean and data type limit identifiers. If you want to associate the data type limit identifiers with the data type names, consider replacing the default identifiers. You can also use command-line parameters to import a header file with the Boolean and data type limit identifier definitions.

Data Type Limit Identifiers

You can control the data type limit identifiers in the generated code by using the command-line parameters in this table.

Data Type Limit	Default Identifier	Command-Line Parameter
8-bit integer maximum	`MAX_int8_T`	`MaxIdInt8`
16-bit integer maximum	`MAX_int16_T`	`MaxIdInt16`
32-bit integer maximum	`MAX_int32_T`	`MaxIdInt32`
8-bit unsigned integer maximum	`MAX_uint8_T`	`MaxIdUint8`
16-bit unsigned integer maximum	`MAX_uint16_T`	`MaxIdUint16`
32-bit unsigned integer maximum	`MAX_uint32_T`	`MaxIdUint32`
8-bit integer minimum	`MIN_int8_T`	`MinIdInt8`
16-bit integer minimum	`MIN_int16_T`	`MinIdInt16`
32-bit integer minimum	`MIN_int32_T`	`MinIdInt32`

For example, to change the default identifiers for the 8-bit integer data limit minimum and maximum to `s4g_S4MIN` and `s4g_S4MAX`, respectively:

```
set_param(gcs,'MinIdInt8','s4g_S4MIN');
set_param(gcs,'MaxIdInt8','s4g_S4MAX')
```

If you do not import a header file, the generated file `rtwtypes.h` defines the 8-bit integer data minimum and maximum identifiers:

```
#define s4g_S4MAX                      ((int8_T)(127))
#define s4g_S4MIN                      ((int8_T)(-128))
```

If you do import a header file defining the data type limit identifiers, the header file is included in `rtwtypes.h`.

Boolean Identifiers

You can control the Boolean identifiers in the generated code by using the command-line parameters in this table. When changing boolean identifiers, you must define `false` to be numerically equivalent to `0`, and `true` to be numerically equivalent to `1`.

Boolean	Default Identifier	Command-Line Parameter
True	`true`	`BooleanTrueId`
False	`false`	`BooleanFalseId`

For example, to change the default Boolean true and false identifiers:

```
set_param(gcs,'BooleanTrueId','bTrue');
set_param(gcs,'BooleanFalseId','bFalse')
```

If you do not import a header file, the generated file `rtwtypes.h` defines the Boolean identifiers:

```
#define bFalse                         (0U)
#define bTrue                          (1U)
```

If you do import a header file defining the Boolean identifiers, the header file is included in `rtwtypes.h`.

Note: When changing boolean identifiers, you must define `false` to be numerically equivalent to `0`, and `true` to be numerically equivalent to `1`.

Boolean and Data Type Limit Identifier Header Files

You can import a header file that defines Boolean and data type limit identifiers using the command-line parameter `TypeLimitIdReplacementHeaderFile`. The header file is included in `rtwtypes.h`. You must use the command-line parameters to specify the Boolean and data type limit identifiers that are included in the imported header file.

For example, if you have a header file `myfile.h` with data type limit definitions, use `TypeLimitIdReplacementHeaderFile` to include the definitions in the generated code:

```
set_param(gcs,'TypeLimitIdReplacementHeaderFile','myfile.h');
```

The generated file `rtwtypes.h` includes `myfile.h`.

```
/* Import type limit identifier replacement definitions. */
#include "myfile.h"
```

Related Examples

- “Conform to Coding Standards by Replacing and Renaming Data Types” on page 21-22
- “Data Type Replacement” on page 21-36

22

Module Packaging Tool (MPT) Data Objects in Embedded Coder

MPT Data Object Properties

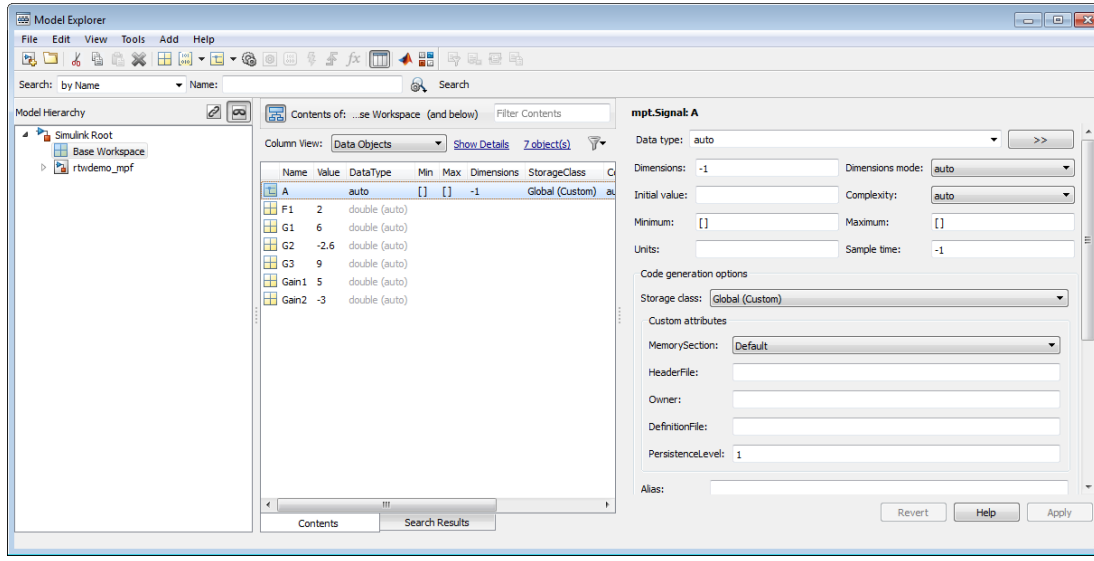
In this section...

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

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

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

In the Properties column, the table lists the properties in the order in which they appear on the Model Explorer.



Parameter and Signal Property Values

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
Both	User object type	`*auto`	Prenamed and predefined property sets that are registered in the `sl_customization.m` file. (See "Register mpt User Object Types" on page 22-16.) This field is active when a user object type is registered. Select `auto` if this field is available but you do not want to apply the properties of a user object type to a selected data object. The fields on the Model Explorer are populated with default values.
		Listed user object type name	Select a user object type name to apply the properties and values that you associated with this name in the `sl_customization.m` file. The fields on the Model Explorer are automatically populated with those values.
Parameter	`Value`	`*0`	The data type and numeric value of the data object. For example, `int8(5)`. The numeric value is used as an initial parameter value in the generated code.
Both	Data type		Used to specify the data type for an `mpt.Signal` data object, but not for an `mpt.Parameter` data object. The data type for an `mpt.Parameter` data object is specified in the **Value** field above. See "About Data Types in Simulink" (Simulink).
Both	Unit	`*null`	Units of measurement of the signal or parameter. (Enter text in this field.)

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
Both	Dimensions	`*-1`	The dimension of the signal or parameter. For a parameter, the dimension is derived from its value.

Both	Complexity	`*auto`
		`real`
		`complex`
			Complexity specifies whether the signal or parameter is a real or complex number. Select `auto` for the code generator to decide. For a parameter, the complexity is derived from its value.

Signal	Sample time	`*-1`	Model or block execution rate.
Signal	Sample mode	`*auto`	Determines how the signal propagates through the model. Select `auto` for the code generator to decide.
		`Sample based`	The signal propagates through the model one sample at a time.
		`Frame based`	The signal propagates through the model in batches of samples.
Both	Minimum	`*0.0`	The minimum value to which the parameter or signal is expected to be bound.
		Number within the minimum range of the parameter or signal. (Based on the data type and resolution of the parameter or signal.)	
Both	Maximum	`*0.0`	Maximum value to which the parameter or signal is expected to be bound. (Enter information using a dialog box.)
	Code generation options		

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
	Storage class		Note that an `auto` selection for a storage class tells the build process to decide how to declare and store the selected parameter or signal.
Both	Default (Custom)		Code generation decides how to declare the data object.
Both	Global (Custom)	`Global (Custom)` is the default storage class for `mpt` data objects.	Specifies that a code generator not place a qualifier in the data object's declaration.
Both	Memory section	`*Default`	**Memory section** allows you to specify storage directives for the data object. `Default` specifies that the code generator not place a type qualifier and `pragma` statement with the data object's declaration.
Parameter		`MemConst`	Places the `const` type qualifier in the declaration.
Both		`MemVolatile`	Places the `volatile` type qualifier in the declaration.
Parameter		`MemConstVolatile`	Places the `const volatile` type qualifier in the declaration.
Both	Header file		Name of the file used to import or export the data object. This file contains the declaration (`extern`) to the data object. Also, you can specify this header filename between the double-quotation or angle-bracket delimiter. You can specify the delimiter with or without the `.h` extension. For example, specify `"object.h"` or `"object"`. For the selected data object, this overrides the general delimiter selection in the

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
			#include file delimiter field on the Configuration Parameters dialog box.
Both	Owner	*Blank	The name of the module that owns this signal or parameter. This is used to help determine the ownership of a definition. For details, see “Ownership Settings” on page 36-106 and the table “Ownership Settings” on page 36-116.
Both	Definition file	*Blank	Name of the file that defines the data object.
		Valid character vector	
Both	Persistence level		The number you specify is relative to **Signal display level** or **Parameter tune level** on the **Code Placement** pane of the Configuration Parameters dialog box. For a signal, allows you to specify whether or not the code generator declares the data object as global data. For a parameter, allows you to specify whether or not the code generator declares the data object as tunable global data. See **Signal display level** and **Parameter tune level** in “Model Configuration Parameters: Code Generation Code Placement”.
Both	Bitfield (Custom)		Embeds Boolean data in a named bit field.
	Struct name		Name of the `struct` into which the object's data will be packed.
Parameter	Const (Custom)		Places the `const` type qualifier in the declaration.
Parameter	Header file		See above.
Parameter	Owner		See above.

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

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
			and places the `#include` in a corresponding file. Assumes external code defines (allocates memory) for the global variable.
Both	Data access	`*Direct`	Allows you to specify whether the identifier that corresponds to the selected data object stores data of a data type (`Direct`) or stores the address of the data (a pointer).
Both		`Pointer`	If you select `Pointer`, the code generator places `*` before the identifier in the generated code.
	Header file		See above.
Both	Struct (Custom)		Embeds data in a named `struct` to encapsulate sets of data.
Both	Struct name		See above.
Signal	GetSet (Custom)		Reads (gets) and writes (sets) data using functions.
Signal	Header file		See above.
Signal	Get function		Specify the Get function.
Signal	Set function		Specify the Set function.
Both	Alias	`*null`	As explained in detail in "Override Data Object Naming Rules" on page 36-18, for a Simulink or `mpt` data object (identifier), specifying a name in the **Alias** field overrides the global naming rule selection you make on the Configuration Parameters dialog box.
		Valid ANSI[a] C/C++ variable name	
Both	`Description`	`*null`	Text description of the parameter or signal. Appears as a comment beside the

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
			signal or parameter's identifier in the generated code.
		Character vector	
Signal	Reusable (Custom)		Allows the code generator to reuse a pair of root I/O signals when you specify the same name and the same custom storage class for both. The custom storage class is either `Reusable (Custom)` or derived from `Reusable (Custom)`.
Signal	Data Scope	`*Auto`	You can specify the scope of symbols code generation generates for a data object of this class by selecting a value for **DataScope**. When you take the default of `Auto`, code generation determines the symbol scope internally. If possible, symbols have `File` scope. Otherwise, they have `Exported` scope.
		`File`	Code generation defines the scope of each symbol as 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, code generation reports an error.
		`Exported`	Code generation exports symbols to external code in the header file specified by the **HeaderFile** field. If a **HeaderFile** is not specified, symbols are exported to external code in *`model`*`.h`.
		`Imported`	Code generation imports symbols from external code in the header file specified by the **HeaderFile** field. If you do not specify a header file, code generation generates an `extern` directive in *`model`*_private.h.

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
Signal	Header file		See above.
Signal	Owner		See above.
Signal	Definition file		See above.

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

`mpt` Package Custom Storage Classes

CSC Name	Purpose	Signals?	Parameters?
`BitField`	Generate a `struct` declaration that embeds Boolean data in named bit fields.	Y	Y
`CompilerFlag`	Supports preprocessor conditionals defined via compiler flag. See "Generate Preprocessor Conditionals for Variant Systems" on page 14-33.	N	Y
`Const`	Generate a constant declaration with the `const` type qualifier.	N	Y
`ConstVolatile`	Generate declaration of volatile constant with the `const volatile` type qualifier.	N	Y
`Default`	The default custom storage class for the `Simulink` package. Export the declaration of data objects to a default generated header file.	Y	Y
`Define`	Generate `#define` directive.	Y	Y
`ExportToFile`	Generate header (`.h`) file, with user-specified name, containing global variable declarations.	Y	Y
`FileScope`	Generate a static qualifier suffix for a variable declaration so that the scope of the variable is limited to the current file.	Y	Y
`GetSet`	Supports specialized function calls to read and write the memory associated with a Data Store Memory block. See "Access Data Through Functions with Custom Storage Class `GetSet`" on page 23-92.	Y	Y

CSC Name	Purpose	Signals?	Parameters?
`Global`	The default custom storage class for the `mpt` package. Generate the declaration and definition of a data object in specified files, and use the specified memory section.	Y	Y
`ImportedDefin`	Supports preprocessor conditionals defined via legacy header file. See “Generate Preprocessor Conditionals for Variant Systems” on page 14-33.	N	Y
`ImportFromFil`	Generate directives to include predefined header files containing global variable declarations.	Y	Y
`Reusable`	Allows the code generator to reuse a pair of root I/O signals when you specify the same name and the same custom storage class for both. The custom storage class is either `Reusable (Custom)` or derived from `Reusable (Custom)`.	Y	N
`Struct`	Generate a `struct` declaration encapsulating parameter or signal object data.	Y	Y
`StructConst`	Generate a `struct` declaration, with a `const` type qualifier, encapsulating parameter object data.	N	Y
`StructVolatil`	Generate a `struct` declaration, with a `volatile` type qualifier, encapsulating parameter or signal object data.	Y	Y
`Volatile`	Use `volatile` type qualifier in declaration.	Y	Y

Examples of Property Value Changes on Generated Code

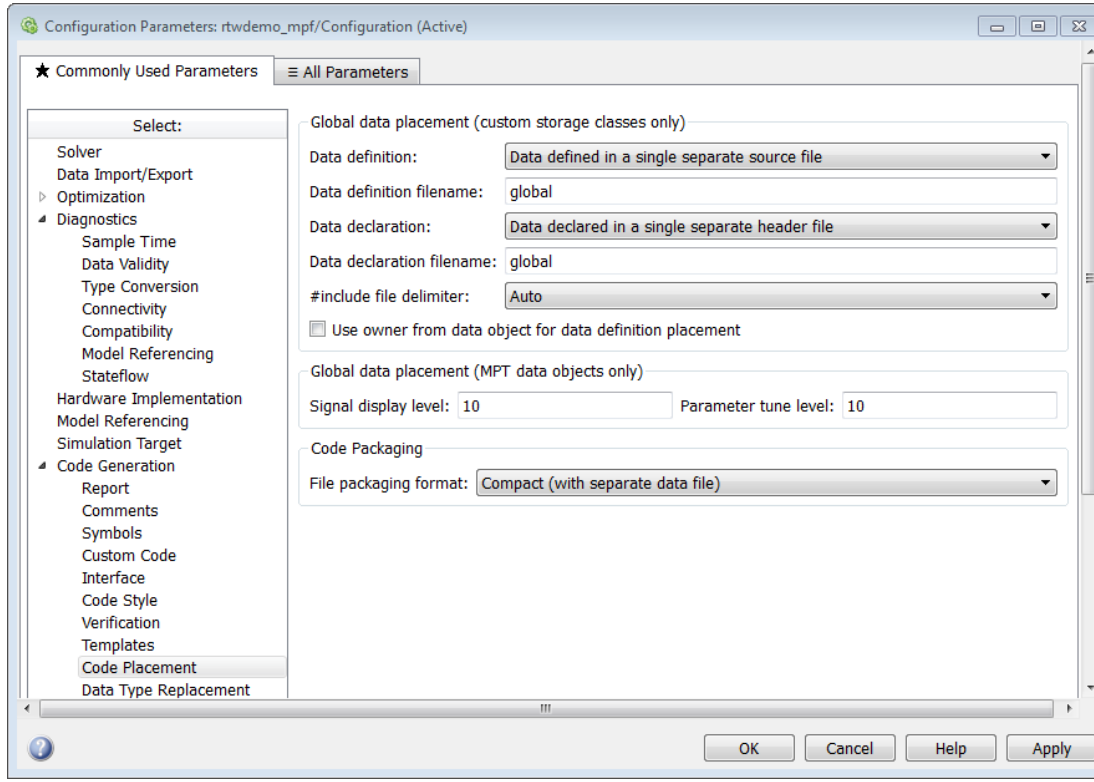
What I noticed when inspecting the .c/.cpp file	Change I made to property value settings	What I noticed after regenerating and reinspecting the file
Example 1: Parameter data objects can be declared or defined as constants. I know that the data object `GAIN` is a parameter. I want this to be declared or defined in the `.c` file as a variable. But I notice that `GAIN` is declared as a constant by the statement `const real_T GAIN = 5.0;`. Also, this statement is in the constant section of the file.	In the Model Explorer, I clicked the data object `GAIN`. I noticed that the property value for its **Memory section** property is set at `MemConst`. I changed this to `Default`.	I notice two differences. One is that now `GAIN` is declared as a variable with the statement `real_T GAIN = 5.0;`. The second difference is that the declaration now is located in the `MemConst` memory section in the `.c` or `.cpp` file.
Example 2: I notice again the declaration of `GAIN` in the `.c` file mentioned in Example 1. It appears as `real_T GAIN = 5.0;`. But I have changed my mind. I want data object `GAIN` to be `#define`.	I changed the **Storage class** selection to `Define (Custom)`.	`GAIN` is not declared in the `.c` file as a `MemConst` parameter. Rather, it is defined as a `#define` macro by the code `#define GAIN 5.0`, and this is located near the top of the `.c` file with the other preprocessor directives.
Example 3: I changed my mind again after doing Example 2. I do want `GAIN` defined using the `#define` preprocessor directive. But I do not want to include the `#define` in this file. I know it exists in another file and I want to reference that file.	On the Model Explorer, I notice that the property value for the **Header file** property is blank. I changed this to `filename.h`. (I chose the ANSI C/C++ double quote mechanism for the `#include`, but could have chosen the angle bracket mechanism.) Also, I must make the user-defined `filename.h` available to the compiler, placing it either in the system path or local directory.	`#define GAIN 5.0` is not present in this `.c` file. Instead, the `#include filename.h` code appears as a preprocessor directive at the top of the file.

What I noticed when inspecting the .c/.cpp file	Change I made to property value settings	What I noticed after regenerating and reinspecting the file
Example 4: I have one more change I want to make. Let us say that we have declared the data object `data_in`, and that its declaration statement in the `.c` file reads `real_T data_in = 0.0;`. I want to replace this statement with an alias in the `.c` file.	In the Model Explorer, I selected the data object `data_in`. I noticed that the **Alias** field is blank. I changed this to `data_in_alias`, which I know is a valid ANSI C/C++ variable name.	The identifier `data_in_alias` now appears in the `.c` file everywhere `data_in` appeared.

Specify Persistence Level for Signals and Parameters

With this procedure, you can control the persistence level of signal and parameter objects associated with a model. Persistence level allows you to make intermediate variables or parameters global during initial development. At the later stages of development, you can use this procedure to remove these signals and parameters for efficiency. Use the **Persistence Level** property of `mpt.Signal` and `mpt.Parameter` data objects. For descriptions of the properties on the Model Explorer, see "MPT Data Object Properties" on page 22-2.

Notice also the **Signal display level** and **Parameter tune level** fields on the **Code Placement** pane of the Configuration Parameters dialog box, as illustrated in the next figure.



The **Signal display level** field allows you to specify whether or not the code generator defines a signal data object as global data in the generated code. The number you specify in this field is relative to the number you specify in the **Persistence level** field. The **Signal display level** number is for `mpt` (module packaging tool) signal data objects in the model. The **Persistence level** number is for a *particular* `mpt` signal data object. If the data object's **Persistence level** is equal to or less than the **Signal display level**, the signal appears in the generated code as global data with the custom attributes that you specified. For example, this would occur if **Persistence level** is `2` and **Signal display level** is `5`.

Otherwise, the code generator automatically determines how the particular signal data object appears in the generated code. Depending on the settings on the **Optimization** pane of the Configuration Parameters dialog box, the signal data object could appear in the code as local data without the custom attributes you specified for that data object. Or,

based on expression folding, the code generator could remove the data object so that it does not appear in the code.

The **Parameter tune level** field allows you to specify whether or not the code generator declares a parameter data object as tunable global data in the generated code.

The number you specify in this field is relative to the number you specify in the **Persistence level** field. The **Parameter tune level** number is for `mpt` parameter data objects in the model. The **Persistence level** number is for a *particular* `mpt` parameter data object. If the data object's **Persistence level** is equal to or less than the **Parameter tune level**, the parameter appears tunable in the generated code with the custom attributes that you specified. For example, this would occur if **Persistence level** is `2` and **Parameter tune level** is `5`.

Otherwise, the parameter is inlined in the generated code, and the code generation settings determine its exact form.

Note that, in the initial stages of development, you might be more concerned about debugging than code size. Or, you might want one or more particular data objects to appear in the code so that you can analyze intermediate calculations of an equation. In this case, you might want to specify the **Parameter tune level** (**Signal display level** for signals) to be higher than **Persistence level** for some `mpt` parameter (or signal) data objects. This results in larger code size, because the code generator defines the parameter (or signal) data objects as global data, which have the custom properties you specified. As you approach production code generation, however, you might have more concern about reducing the size of the code and less need for debugging or intermediate analyses. In this stage of the tradeoff, you could make the **Parameter tune level** (**Signal display level** for signals) greater than **Persistence level** for one or more data objects, generate code and observe the results. Repeat until satisfied.

1. With the model open, in the Configuration Parameters dialog box, select **Code Generation** > **Code Placement**.
2. Type the desired number in the **Signal display level** or **Parameter tune level** field, and click **Apply**.
3. In the Model Explorer, type the desired number in the **Persistence** field for the selected signal or parameter, and click **Apply**.
4. Save the model and generate code.

Register mpt User Object Types

- “Introduction” on page 22-17

Introduction

Embedded Coder allows you to create custom `mpt` object types and specify properties and property values to be associated with them. Once created, a user object type can be applied to data objects displayed in Model Explorer. When you apply a user object type to a data object, by selecting a type name in the **User object type** pull-down list in Model Explorer, the data object is automatically populated with the properties and property values that you specified for the user object type.

To register `mpt` user object type customizations, use the Simulink customization file `sl_customization.m`. This file is a mechanism that allows you to use MATLAB code to perform customizations of the standard Simulink user interface. The Simulink software reads the `sl_customization.m` file, if present on the MATLAB path, when it starts and the customizations specified in the file are applied to the Simulink session. For more information on the `sl_customization.m` customization file, see "Registering Customizations" (Simulink).

Register mpt User Object Types Using sl_customization.m

To register `mpt` user object type customizations, you create an instance of `sl_customization.m` and include it on the MATLAB path of the Simulink installation that you want to customize. The `sl_customization` function accepts one argument: a handle to a customization manager object. For example,

```
function sl_customization(cm)
```

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

```
hObj = cm.slDataObjectCustomizer;
```

You then invoke methods to register your customizations. The customization manager object includes the following methods for registering `mpt` user object type customizations:

- `addMPTObjectType(hObj, objectTypeName, classtype, propName1, propValue1, propName2, propValue2, ...)`

  `addMPTObjectType(hObj, objectTypeName, classtype, {propName1, propName2, ...}, {propValue1, propValue2, ...})`

Registers the specified user object type, along with specified values for object properties, and adds the object type to the top of the user object type list, as displayed in the **User object type** pull-down list in the Model Explorer.

- `objectTypeName` — Name of the user object type
- `classType` — Class to which the user object type applies: `'Signal'`, `'Parameter'`, or `'Both'`
- `propName` — Name of a property of an `mpt` or `mpt`-derived data object to be populated with a corresponding `propValue` when the registered user object type is selected
- `propValue` — Specifies the value for a corresponding `propName`

- `moveMPTObjectTypeToTop(hObj, objectTypeName)`

  Moves the specified user object type to the top of the user object type list, as displayed in the **User object type** pull-down list in the Model Explorer.

- `moveMPTObjectTypeToEnd(hObj, objectTypeName)`

  Moves the specified user object type to the end of the user object type list, as displayed in the **User object type** pull-down list in the Model Explorer.

- `removeMPTObjectType(hObj, objectTypeName)`

  Removes the specified user object type from the user object type list.

Your instance of the `sl_customization` function should use these methods to register `mpt` object type customizations for your Simulink installation.

The Simulink software reads the `sl_customization.m` file when it starts. If you subsequently change the file, to use the changes, you must restart your MATLAB session.

mpt User Object Type Customization Using sl_customization.m

The `sl_customization.m` file shown in sl_customization.m for mpt Object Type Customizations uses the `addMPTObjectType` method to register the user signal types `EngineType` and `FuelType` for `mpt` objects.

sl_customization.m for mpt Object Type Customizations

```
function sl_customization(cm)
```

```
% Register user customizations

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

% Add commonly used signal types
hObj.addMPTObjectType(...
    'EngineType','Signal',...
    'DataType', 'uint8',...
    'Min', 0,...
    'Max', 255,...
    'Unit','m/s');

hObj.addMPTObjectType(...
    'FuelType','Signal',...
    'DataType', 'int16',...
    'Min', -12,...
    'Max', 3000,...
    'Unit','mg/hr');

end
```

If you include the above file on the MATLAB path of the Simulink installation that you want to customize, the specified customizations will appear in Model Explorer. For example, you could view the customizations as follows:

1. Start a MATLAB session.
2. Open Model Explorer, for example, by entering the MATLAB command `daexplr`.
3. Select **Base Workspace**.
4. Add an `mpt` signal, for example, by selecting **Add** > **Add Custom**.
5. In the right-hand pane display for the added `mpt` signal, examine the **User object type** drop-down list, noting the impact of the changes specified in sl_customization.m for mpt Object Type Customizations.
6. From the **User object type** drop-down list, select one of the registered user signal types, for example, `FuelType`, and verify that the displayed settings are consistent with the arguments specified to the `addMPTObjectType` method in `sl_customization.m`.

Custom Storage Classes in Embedded Coder

Introduction to Custom Storage Classes

In this section...

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

The code generator defines four built-in storage classes for use with system target files: `Auto`, `ExportedGlobal`, `ImportedExtern`, and `ImportedExternPointer`. These storage classes provide limited control over the form of the code generated for references to the data. For example, data of storage class `Auto` is typically declared and accessed as an element of a structure, while data of storage class `ExportedGlobal` is declared and accessed as unstructured global variables. For information about built-in storage classes, see "Control Signals and States in Code by Applying Storage Classes" (Simulink Coder) and "Block Parameter Representation in the Generated Code" (Simulink Coder).

If the built-in storage classes do not provide data representation required by your application, you can define *custom storage classes* (CSCs). Embedded Coder CSCs extend the built-in storage classes provided by Simulink Coder. CSCs can provide application-specific control over the constructs required to represent data in an embedded algorithm. For example, you can use CSCs to:

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

Custom storage classes affect only code generated for ERT targets. When **Configuration Parameters > Code Generation > Target Selection > System target file** specifies a GRT target, the names of custom storage classes sometimes appear in dialog boxes, but selecting a CSC is functionally the same as selecting `Auto`. For information about ERT and GRT targets, see "Compare System Target File Support" (Simulink Coder).

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 "Create and Edit Memory Section Definitions" on page 23-52 for details about using the Custom Storage Class designer to define memory sections. While memory sections are often used with data in custom storage classes, they can also be used with various other constructs. See "Control Data and Function Placement in Memory by Inserting Pragmas" on page 27-2 for more information about using memory sections with custom storage classes, and complete information about using memory sections with other constructs.

Custom Storage Classes and Data Class Packages

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). A custom storage class is available only to data classes that are defined by the associated package.

You cannot add or change CSCs associated with built-in packages and classes, but you can create your own packages and subclasses, then associate customized CSCs with those packages. To create your own packages and custom storage classes, see "Design Custom Storage Classes and Memory Sections" on page 23-34.

Custom Storage Class Examples

These examples show Custom Storage Class capabilities:

"Configure Data Interface by Applying Custom Storage Classes" — Shows how custom storage classes can support data-object-driven modeling

rtwdemo_cscpredef — Shows predefined custom storage classes and embedded signal objects

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

Click the links above, or type the name in the MATLAB Command Window.

Related Examples

- "Simulink Package Custom Storage Classes" on page 23-5
- "Control Data Representation by Applying Custom Storage Classes" on page 23-58
- "Exchange and Reuse Parameter Data Between Generated Code and Existing Code" on page 23-11
- "Signal Representation in Generated Code" (Simulink Coder)
- "Block Parameter Representation in the Generated Code" (Simulink Coder)

Simulink Package Custom Storage Classes

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

- Extend the storage classes provided by Simulink Coder
- Are functionally the same as if you had defined them yourself using the CSC Designer

You cannot change the CSCs built into the `Simulink` package, but you can subclass the package and add CSCs to the subclass, following the steps in "Resources for Defining Custom Storage Classes" on page 23-34.

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

The code generator defines four built-in storage classes for use with targets: `Auto`, `ExportedGlobal`, `ImportedExtern`, and `ImportedExternPointer`. These storage classes provide limited control over the form of the code generated for references to the data. For example, data of storage class `Auto` is typically declared and accessed as an element of a structure, while data of storage class `ExportedGlobal` is declared and accessed as unstructured global variables. For information about built-in storage classes, see "Control Signals and States in Code by Applying Storage Classes" (Simulink Coder) and "Block Parameter Representation in the Generated Code" (Simulink Coder).

CSC Name	Purpose	Signals?	Parameters?
`BitField`	Generate a `struct` declaration that embeds Boolean data in named bit fields. For an example, see "Bitfields" on page 13-95.	Y	Y
`CompilerFlag`	Supports preprocessor conditionals defined via compiler flag. See "Generate Preprocessor Conditionals for Variant Systems" on page 14-33. To specify the compiler option, use the model configuration parameter	N	Y

CSC Name	Purpose	Signals?	Parameters?
	Configuration Parameters > **Code Generation** > **Custom Code** > **Additional build information** > **Defines**. See Code Generation Pane: Custom Code: Additional Build Information: Defines (Simulink Coder).		
`Const`	Generate a constant declaration with the `const` type qualifier.	N	Y
`ConstVolatile`	Generate declaration of volatile constant with the `const volatile` type qualifier. For an example, see "Type Qualifiers" on page 13-15.	N	Y
`Default`	`Default` is a placeholder CSC that the code generator assigns to the `CoderInfo.CustomStorageClass` property of signal and parameter objects when they are created. The signal, state, or parameter appears in the generated code as a global variable.	Y	Y
`Define`	Generate `#define` directive. The generated code defines the macro value. For an example, see "Macro Definitions (#define)" on page 13-77. Also supports generation of preprocessor conditionals for variant systems. See "Generate Preprocessor Conditionals for Variant Systems" on page 14-33.	Y	Y
`ExportToFile`	Generate header (`.h`) file, with user-specified name, containing global variable declarations.	Y	Y

CSC Name	Purpose	Signals?	Parameters?
`FileScope`	Generate a static qualifier suffix for a variable declaration so that the scope of the variable is limited to the current file.	Y	Y
`GetSet`	Supports specialized function calls to read and write memory. You can use this custom storage class with data stores. For examples, see "Access Data Through Functions with Custom Storage Class `GetSet`" on page 23-92.	Y	Y
`ImportedDefine`	Generate `#define` directive. You supply the macro definition in a legacy header file. For an example, see "Macro Definitions (#define)" on page 13-77. Also supports preprocessor conditionals, for variant systems, defined via legacy header file. See "Generate Preprocessor Conditionals for Variant Systems" on page 14-33.	N	Y
`ImportFromFile`	Generate directives to include predefined header files containing global variable declarations.	Y	Y

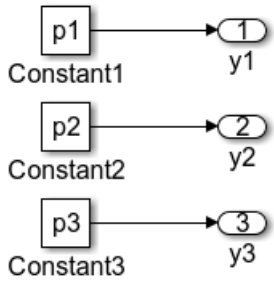
CSC Name	Purpose	Signals?	Parameters?
`Reusable`	Allows the code generator to reuse a pair of root I/O signals when you specify the same name and the same custom storage class for both. The custom storage class is either `Reusable (Custom)` or derived from `Reusable (Custom)`. For an example, see "Specify Buffer Reuse for Multiple Signals in a Path" on page 55-19. You can apply this storage class only to `Simulink.Signal` objects. For example, you cannot apply the storage class by using a Signal Properties dialog box.	Y	N
`Struct`	Generate a `struct` declaration encapsulating parameter or signal object data. For examples, see "Organize Parameter Data into a Structure by Using the `Struct` Custom Storage Class" on page 23-8 and "Structures of Signals" on page 13-87.	Y	Y
`Volatile`	Use `volatile` type qualifier in declaration.	Y	Y

Organize Parameter Data into a Structure by Using the `Struct` Custom Storage Class

This example shows how to use the `Struct` custom storage class to organize block parameter values into a structure in the generated code.

To create a structure of parameter data in the generated code, consider creating a corresponding structure in Simulink. See "Organize Block Parameter Values into Structures in the Generated Code" on page 19-97.

1. Create the `ex_struct_param` model with three Constant blocks and three Outport blocks.



2. Create a data object for each parameter, `p1`, `p2`, and `p3`. At the MATLAB command line, enter:

```
p1 = Simulink.Parameter;
p2 = Simulink.Parameter;
p3 = Simulink.Parameter;
```

3. In the base workspace, double-click one of the parameter data objects to open the `Simulink.Parameter` dialog box.
4. Specify a **Value** parameter for each parameter object.
5. Specify the **Storage class** parameter as `Struct` for each parameter object.
6. In the **Custom Attributes** section, specify the **StructName** as `my_struct`. Click **Apply** and **OK**.
7. Press **Ctrl+B** to generate code.

The generated code includes the `typedef` definition for a structure, which is declared in the `ex_struct_param_types.h` file.

```
/* Type definition for custom storage class: Struct */
   typedef struct my_struct_tag {
      real_T p1;
      real_T p2;
      real_T p3;
   } my_struct_type;
```

The generated code also includes the declaration of `my_struct` in `ex_struct_param.c`.

```
/* Definition for custom storage class: Struct */
my_struct_type my_struct = {
     /* p1 */
     1.0,

     /* p2 */
```

```
    2.0,

    /* p3 */
    3.0
};
```

Related Examples

- “Control Data Code by Creating Custom Storage Class” on page 23-73
- “Control Data Representation by Applying Custom Storage Classes” on page 23-58
- “Generate Code with Custom Storage Classes” on page 23-67
- “Design Custom Storage Classes and Memory Sections” on page 23-34
- “Data Objects” (Simulink)
- “Define Advanced Custom Storage Classes Types” on page 23-78

Exchange and Reuse Parameter Data Between Generated Code and Existing Code

Blocks have numeric parameters that determine how they calculate output values, for example, the **Gain** parameter of a Gain block. In the generated code, you can configure block parameters to appear as parameter data, which include global variables and the formal parameters of a function (see "Block Parameter Representation in the Generated Code" on page 19-47).

When you integrate the generated code with your existing custom code, you can configure the generated code to reuse parameter data that your custom code defines. You can also generate code that defines parameter data for your custom code to use. For example, you can configure a **Gain** parameter to refer to a `Simulink.Parameter` object, which you can configure to appear in the generated code as a global variable. You can then use that variable in your existing code.

When you generate code that defines (allocates memory for and initializes) parameter data, the generated code exports that data. When your custom code defines parameter data, the generated code imports that data.

To export or import global parameter data:

1. Create a `Simulink.Parameter` object to represent the parameter data. Use the `Value` property of the object to store the parameter value for simulation in Simulink and for generation of an exported definition.

   To package lookup table data according to the ASAP2 or AUTOSAR standards (for example, STD_AXIS or MAP), use Simulink.LookupTable or Simulink.Breakpoint objects instead of `Simulink.Parameter` objects.
2. Optionally, specify the data type (for example, `int32`) of the parameter data by configuring the parameter object. You can generate code that uses custom data types (`typedef`) from your code. You can also represent structures, enumerations, and Boolean data. If you have Fixed-Point Designer, you can represent fixed-point data types.
3. Apply a storage class or custom storage class to the parameter object. Use the storage class to control the *scope* of the data. The data scope indicates whether the generated code imports or exports the data definition.

   For most storage classes, the generated code algorithm accesses the data through global reference.

For an example, see “Reuse Parameter Data from Custom Code in the Generated Code” on page 23-17.

Control Data Scope

To specify whether global parameter data are imported or exported, use storage classes and custom storage classes . Typically, storage classes that import data have the word `Import` in the storage class name, for example `ImportedExternPointer`.

For more information about the storage classes that are built into the code generator, see “Override Default Parameter Behavior by Creating Global Variables in the Generated Code” on page 19-49. For more information about using and creating custom storage classes, see “Introduction to Custom Storage Classes” on page 23-2 and “Simulink Package Custom Storage Classes” on page 23-5.

Control File Placement of Exported Parameter Data

When you export parameter data from the generated code by using storage classes or custom storage classes, the code generator creates an `extern` declaration. By default, this declaration typically appears in the generated header file *`model`*`.h`. You can include (`#include`) this header file in your code.

By default, the definition (memory allocation) and static initialization of exported parameter data typically appear in *`model`*`.c`.

You can control the file placement of the declarations and definitions to:

- Create separate object files that store only global parameter data.
- Modularize the generated code by organizing declarations into separate files.

To control the default file placement for parameter data that use custom storage classes, use the model configuration parameters **Configuration Parameters > Code Generation > Code Placement > Data definition** and **Data declaration** (see “Data definition” and “Data declaration”).

To control file placement for individual parameter data items (parameter objects), use the custom storage class `ExportToFile` or create your own similar custom storage class. For an example, see “Definition, Initialization, and Declaration of Parameter Data” on page 13-8.

For more information about controlling file placement of declarations and definitions, see “Manage Placement of Data Definitions and Declarations” on page 36-100.

Customize and Control Parameter Data Types

When your code uses `typedef` statements to define custom data types as aliases of integer and floating-point data types, such as `typedef int my_int;`, you can generate code that uses these different data types.

1. Create a `Simulink.AliasType` or `Simulink.NumericType` object.
2. Use the `DataScope` and `HeaderFile` properties of the object to import the type definition from your code.
3. Use the object as the data type of block parameters and parameter objects.

When you share parameter data between the generated code and your custom code, if you leave the data type of block parameters and `Simulink.Parameter` objects at the default settings (typically `Inherit: Inherit via internal rule` and `auto`), Simulink chooses the parameter and object data types. In some cases, when you make changes to the model (for instance, by changing the data types of signals), Simulink chooses different data types. Unless you modify your custom code so that it uses the new parameter data type, this change can cause compiler errors when you integrate the generated code with your custom code. To prevent the errors, you can explicitly specify the data type of the parameter object by using the `DataType` property. The corresponding global variable in the generated code uses the data type that you specify. For an example, see "Reuse Parameter Data from Custom Code in the Generated Code" on page 23-17 .

For more information about using data type objects to set data types in a model, see "Create and Apply User-Defined Data Types" on page 21-9 and "Conform to Coding Standards by Replacing and Renaming Data Types" on page 21-22. For more information about controlling parameter data types, see "Parameter Data Types in the Generated Code" on page 19-79.

To replace data type names throughout a model by default, consider using data type replacement. See "Data Type Replacement" on page 21-36.

Enumerated Parameters

You can export or import the definition of an enumerated data type to the generated code. For instance, use this technique to import parameter data that uses an enumerated type defined by your code.

To create a Simulink representation of an enumeration that your existing C code defines, use the `Simulink.importExternalCTypes` function. For an example that shows

how to export enumerated parameter data, see “Enumeration” on page 13-24. For more information, see “Use Enumerated Data in Generated Code” on page 19-22.

Pass Imported Parameter Data to Generated Algorithm as Arguments

You can generate a reusable algorithm that accepts parameter data through formal parameters of the model entry-point functions. You can then pass a different parameter value to each function call in the generated code or in your code.

To parameterize reusable referenced models and subsystems, use model arguments and mask parameters. For information, see “Parameter Interfaces for Reusable Components” (Simulink).

When you write custom code that calls the generated model `step` function multiple times, you can configure the model to generate a reentrant function that accepts signal, state, and parameter data as formal parameters. Set the model configuration parameter **Code interface packaging** to `Reusable function` (see “Code interface packaging” (Simulink Coder)).

When you use this technique, to pass parameter data through the reentrant interface, you must configure the parameter data to appear in the generated code as fields of the global parameter structure.

- Set the model configuration parameter **Default parameter behavior** (see “Default parameter behavior” (Simulink)) to `Tunable`. By default, block parameters appear in the generated code as fields of the parameter structure.

  Use this technique for rapid prototyping.
- In the model, create parameter objects (for example, `Simulink.Parameter` or `Simulink.LookupTable`) to set block parameter values. Optionally, use structures to group multiple parameter values into a single object (see “Organize Related Block Parameter Definitions in Structures” (Simulink)). Apply the storage class `SimulinkGlobal` to the parameter objects. These parameter objects appear in the generated code as fields of the parameter structure.

  Use this technique when you set **Default parameter behavior** to `Inlined` for production code generation.

For more information about **Code interface packaging** and reentrancy, see “Generate Reentrant Code from Top-Level Models” on page 34-20. For information about

creating instance-specific data in your custom code, see “Modify Static Main to Allocate and Access Model Instance Data” on page 49-14.

Considerations for Other Modeling Goals

Goal	Considerations and More Information
Use multidimensional parameters (arrays)	The generated code defines and accesses multidimensional data, including matrices, as column-major serialized vectors. If your custom code uses a different format, consider using alternative techniques to integrate the generated code. See “Code Generation of Matrices and Arrays” on page 33-76.
Use parameter structures	You can export or import a structure of parameter values by creating a `Simulink.Bus` object to represent the structure type. Use the bus object as the data type of a parameter object. You can also use custom storage classes for greater modeling flexibility. See “Organize Block Parameter Values into Structures in the Generated Code” on page 19-97.
Use macros (`#define`)	To export a macro to your custom code, you can use the custom storage class `Define`. To import a macro, use `ImportedDefine`. With macros, you can reuse a parameter value in multiple locations in an algorithm and change the parameter value between code compilations without consuming memory to store the value. Typically, macros represent engineering constants that you do not expect to change during code execution. For examples, see “Macro Definitions (#define)” on page 13-77.
Generate code that imports parameter data through custom functions	When your code contains functions, such as device drivers, that return parameter data, you can use custom storage classes to generate code that calls the functions. See “Access Data Through Functions with Custom Storage Class `GetSet`” on page 23-92.
Use storage type qualifiers such as `const` and `volatile`	When you import parameter data from your code, you can generate code that matches the storage type qualifiers that your code applies to the data. Use custom storage classes and memory sections. See “Type Qualifiers” on page 13-15.
Generate code comments that describe attributes of data including physical units, real-	Generating these comments can help you match data interfaces while handwriting integration code. See “Add Custom Comments for Variables in the Generated Code” on page 36-5.

Goal	Considerations and More Information
world initial value, and data type	
Use Simulink to simulate an external executable	You can use SIL, PIL, and external mode simulations to connect your model to the corresponding generated executable for simulation. When you import parameter data from your custom code: • At the time that you begin an external mode simulation, the external executable uses the value that your code uses to initialize the parameter data. However, when you change the corresponding value in Simulink during the simulation (for example by modifying the `Value` property of the corresponding parameter object), Simulink downloads the new value to the executable. • SIL and PIL simulations do not import the parameter value from your code. Instead, the simulations use the parameter value from Simulink.

Related Examples

- "Reuse Parameter Data from Custom Code in the Generated Code" on page 23-17
- "Import Parameter Data with Conditionally Compiled Dimension Length" on page 23-22
- "Exchange Data Between External C/C++ Code and Simulink Model or Generated Code" on page 39-86
- "Configure Generated Code According to Interface Control Document" on page 23-112
- "Create Tunable Calibration Parameter in the Generated Code" on page 19-60
- "Introduction to Custom Storage Classes" on page 23-2
- "Block Parameter Representation in the Generated Code" on page 19-47
- "Data Objects" (Simulink)

Reuse Parameter Data from Custom Code in the Generated Code

This example shows how to generate code that imports a parameter value from your external, custom code.

Create Custom Code Files

Suppose your custom code defines a vector parameter `myGains` with three elements. Save the definition in your current folder in a file called `ex_vector_import_src.c`.

```
#include "ex_vector_import_decs.h"

my_int8 myGains[3] = {
    2,
    4,
    6
};
```

Save the declaration in your current folder in a file called `ex_vector_import_decs.h`.

```
#include "ex_vector_import_cust_types.h"

extern my_int8 myGains[3];
```

Save the custom data type definition `my_int8` in your current folder in a file called `ex_vector_import_cust_types.h`.

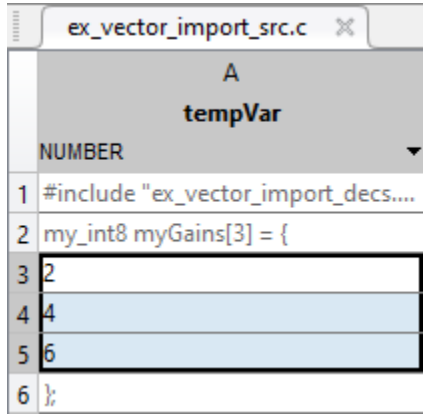
```
typedef signed char my_int8;
```

Import Parameter Value for Simulation

In your current folder, right-click the file `ex_vector_import_src.c` and select **Import Data**.

In the Import dialog box, set the name of the generated MATLAB variable to `tempVar`.

Select only the parameter values (`2`, `4`, and `6`) to import.



Import the data by clicking the green check mark. The MATLAB variable `tempVar` appears in the base workspace.

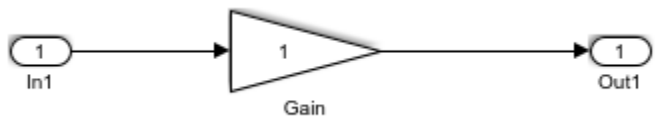
Alternatively, use the command prompt to manually create `tempVar`.

```
tempVar = [2;4;6];
```

Create and Configure Model

Create the model `ex_vector_import`.

```
open_system('ex_vector_import')
```



In the Gain block dialog box, on the **Parameter Attributes** tab, set **Parameter data type** to `Inherit: Inherit from 'Gain'`.

On the **Main** tab, set **Gain** to `myGains`. Click **Apply**.

Click the action button next to the parameter value. Select **Create Variable**.

In the Create New Data dialog box, set **Value** to `Simulink.Parameter` and click **Create**.

In the myGains dialog box, set these property values and click **OK**:

- **Data type** to `my_int8`
- **Storage class** to `ImportFromFile`
- **HeaderFile** to `ex_vector_import_decs.h`

Alternatively, use these commands at the command prompt to create the object and set the property values:

```
myGains = Simulink.Parameter;
set_param('ex_vector_import/Gain','Gain','myGains',...
    'ParamDataTypeStr','Inherit: Inherit from ''Gain''')
myGains.DataType = 'my_int8';
myGains.CoderInfo.StorageClass = 'Custom';
myGains.CoderInfo.CustomStorageClass = 'ImportFromFile';
myGains.CoderInfo.CustomAttributes.HeaderFile = 'ex_vector_import_decs.h';
```

At the command prompt, set the `Value` property by using the value of `tempVar`.

```
myGains.Value = tempVar;
```

At the command prompt, create a `Simulink.AliasType` object to represent your custom data type `my_int8`. Set the `DataScope` and `HeaderFile` properties to import the type definition from your custom code.

```
my_int8 = Simulink.AliasType('int8');
my_int8.DataScope = 'Imported';
my_int8.HeaderFile = 'ex_vector_import_cust_types.h';
```

Set **Configuration Parameters > Code Generation > Custom Code > Additional build information > Source files** to `ex_vector_import_src.c`.

```
set_param('ex_vector_import','CustomSource','ex_vector_import_src.c')
```

Generate and Inspect Code

Generate code from the model.

```
rtwbuild('ex_vector_import')
```

```
### Starting build procedure for model: ex_vector_import
### Successful completion of build procedure for model: ex_vector_import
```

The generated file `ex_vector_import.h` includes the custom header files `ex_vector_import_decs.h` and `ex_vector_import_cust_types.h`, which contain the parameter variable declaration (`myGains`) and custom type definition (`my_int8`).

```
file = fullfile('ex_vector_import_ert_rtw','ex_vector_import.h');
rtwdemodbtype(file,'/* Includes for objects with custom storage classes. */',...
    '#include "ex_vector_import_cust_types.h"',1,1)
```

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

The generated code algorithm in the model `step` function in the generated file `ex_vector_import.c` uses `myGains` for calculations.

```
file = fullfile('ex_vector_import_ert_rtw','ex_vector_import.c');
rtwdemodbtype(file,'/* Model step function */','/* Model initialize function */',1,0)
```

```
/* Model step function */
void ex_vector_import_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Gain: '<Root>/Gain'
   *  Inport: '<Root>/In1'
   */
  rtY.Out1[0] = (real_T)myGains[0] * rtU.In1;
  rtY.Out1[1] = (real_T)myGains[1] * rtU.In1;
  rtY.Out1[2] = (real_T)myGains[2] * rtU.In1;
}
```

The generated code does not define (allocate memory for) or initialize the global variable `myGains` because the data scope of the corresponding parameter object is imported.

When you simulate the model in Simulink, the model uses the value stored in the `Value` property of the parameter object. However, if you use external mode simulation, the external executable begins the simulation by using the value from your code. See "Considerations for Other Modeling Goals".

Related Examples

- "Exchange and Reuse Parameter Data Between Generated Code and Existing Code" on page 23-11

- "Import Parameter Data with Conditionally Compiled Dimension Length" on page 23-22

Import Parameter Data with Conditionally Compiled Dimension Length

Suppose your custom code conditionally allocates memory for and initializes lookup table and breakpoint set data based on a dimension length that you specify as a `#define` macro. This example shows how to generate code that imports this external global data.

Create Custom Code Files

Save the definition of the breakpoint set data `T1Break` and lookup table data `T1Data` in your current folder in a file called `ex_vec_symdim_src.c`. These global variables have either 9 or 11 elements depending on the value of the macro `bpLen`.

```
#include "ex_vec_symdim_decs.h"

#if bpLen == 11
double T1Break[bpLen] = {
    -5.0,
    -4.0,
    -3.0,
    -2.0,
    -1.0,
    0.0,
    1.0,
    2.0,
    3.0,
    4.0,
    5.0
} ;

double T1Data[bpLen] = {
    -1.0,
    -0.99,
    -0.98,
    -0.96,
    -0.76,
    0.0,
    0.76,
    0.96,
    0.98,
    0.99,
    1.0
```

```
} ;
#endif

#if bpLen == 9
double T1Break[bpLen] = {
    -4.0,
    -3.0,
    -2.0,
    -1.0,
    0.0,
    1.0,
    2.0,
    3.0,
    4.0
} ;

double T1Data[bpLen] = {
    -0.99,
    -0.98,
    -0.96,
    -0.76,
    0.0,
    0.76,
    0.96,
    0.98,
    0.99
} ;
#endif
```

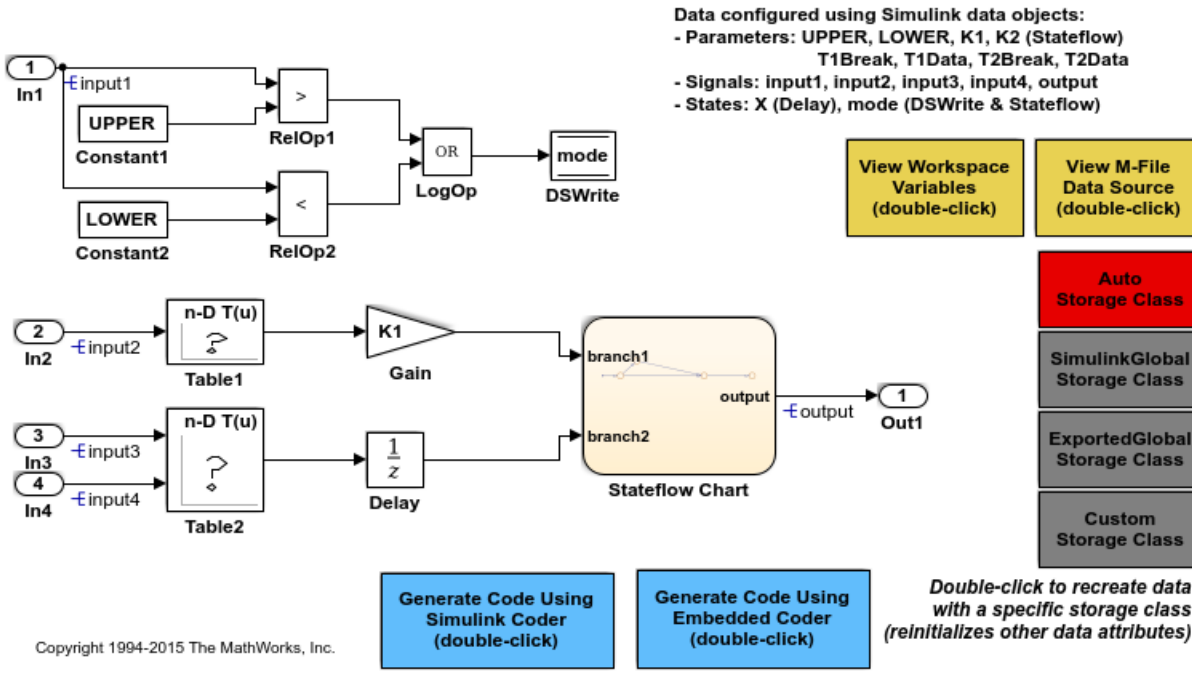
Save the declarations of the variables and the definition of the macro in your current folder in a file called `ex_vec_symdim_decs.h`.

```
#define bpLen 11

extern double T1Break[bpLen];
extern double T1Data[bpLen];
```

Explore and Configure Example Model

Open the example model `rtwdemo_advsc`.

```
open_system('rtwdemo_advsc')
```



Open the Table1 block dialog box. The block refers to `Simulink.Parameter` objects, `T1Data` and `T1Break`, in the base workspace. These objects store the lookup table and breakpoint set data with 11 elements.

At the command prompt, configure the objects to import the data definitions from your custom code.

```
T1Data.CoderInfo.StorageClass = 'Custom';
T1Data.CoderInfo.CustomStorageClass = 'ImportFromFile';
T1Data.CoderInfo.CustomAttributes.HeaderFile = 'ex_vec_symdim_decs.h';

T1Break.CoderInfo.StorageClass = 'Custom';
T1Break.CoderInfo.CustomStorageClass = 'ImportFromFile';
T1Break.CoderInfo.CustomAttributes.HeaderFile = 'ex_vec_symdim_decs.h';
```

At the command prompt, create a `Simulink.Parameter` object to represent the custom macro `bpLen`.

```
bpLen = Simulink.Parameter(11);
```

```
bpLen.Min = 9;
bpLen.Max = 11;
bpLen.DataType = 'int32';
bpLen.CoderInfo.StorageClass = 'Custom';
bpLen.CoderInfo.CustomStorageClass = 'ImportedDefine';
bpLen.CoderInfo.CustomAttributes.HeaderFile = 'ex_vec_symdim_decs.h';
```

Use `bpLen` to set the dimensions of the lookup table and breakpoint set data. Configure the model to enable symbolic dimensions by selecting the configuration parameter **Allow symbolic dimension specification**.

```
T1Data.Dimensions = '[1 bpLen]';
T1Break.Dimensions = '[1 bpLen]';
set_param('rtwdemo_advsc','AllowSymbolicDim','on')
```

Set **Configuration Parameters > Code Generation > Custom Code > Additional build information > Source files** to `ex_vec_symdim_src.c`.

```
set_param('rtwdemo_advsc','CustomSource','ex_vec_symdim_src.c')
```

Generate and Inspect Code

Generate code from the model.

```
rtwbuild('rtwdemo_advsc')
```

```
### Starting build procedure for model: rtwdemo_advsc
### Successful completion of build procedure for model: rtwdemo_advsc
```

The generated code algorithm is in the model `step` function in the generated file `rtwdemo_advsc.c`. The algorithm passes `T1Break`, `T1Data`, and `bpLen` as argument values to the function that performs the table lookup. In this case, `bpLen` controls the upper bound of the binary search that the function uses.

```
file = fullfile('rtwdemo_advsc_ert_rtw','rtwdemo_advsc.c');
rtwdemodbtype(file,' look1_binlc','bpLen - 1U',1,1)
```

```
    rtwdemo_advsc_DWork.X = look1_binlc(rtwdemo_advsc_U.input2, (&(T1Break[0])),
      (&(T1Data[0])), bpLen - 1U) * 2.0;
```

For more information about symbolic dimensions, see “Implement Dimension Variants for Array Sizes in Generated Code”.

Related Examples

- “Exchange and Reuse Parameter Data Between Generated Code and Existing Code” on page 23-11
- “Reuse Parameter Data from Custom Code in the Generated Code” on page 23-17

Access Structured Data Through a Pointer That External Code Defines

This example shows how to generate code that uses global data that some handwritten code defines. In the handwritten code, a pointer variable points to one of two structure variables that contain parameter data. A handwritten function switches the pointer between the two structures. The generated code accesses the parameter data by dereferencing the pointer variable.

Explore Custom Code

Open the example source file rtwdemo_importstruct_user.c. The code defines a structure variable `ReferenceStruct` as constant (`const`) data and statically initializes each field.

```
/* Constant default data struct: ReferenceStruct */
const DataStruct_type ReferenceStruct =
{
  11,   /* OFFSET */
  2     /* GAIN */
};
```

The code defines another structure variable, `WorkingStruct`, as volatile (`volatile`) data.

```
/* Volatile data struct: WorkingStruct */
volatile DataStruct_type WorkingStruct;
```

The code defines a function that copies the field values from `ReferenceStruct` to `WorkingStruct`.

```
/* Function to initialize WorkingStruct with data from ReferenceStruct */
void Init_WorkingStruct(void)
{
      memcpy((void*)&WorkingStruct, &ReferenceStruct, sizeof(ReferenceStruct));
}
```

The code defines `StructPointer`, which is a `const volatile` pointer to a structure. The code initializes the pointer to the address of `ReferenceStruct`.

```
/* Create pointer to the default data struct, e.g. ReferenceStruct */
const volatile DataStruct_type *StructPointer = &ReferenceStruct;
```

Finally, the code defines a function that can dynamically set `StructPointer` to point to either `ReferenceStruct` or `WorkingStruct`.

```
/* Function to switch between structures */
void SwitchStructPointer(Dataset_T Dataset)
{
    switch (Dataset)
    {
        case Working:
          StructPointer = &WorkingStruct;
          break;
        default:
          StructPointer = &ReferenceStruct;
    }
}
```

The example header file rtwdemo_importstruct_user.h defines the enumeration `Dataset_T` and the structure type `Datastruct_type`. The file includes (`#include`) the built-in Simulink® Coder™ header file `rtwtypes.h`, which defines (`typedef`) Simulink Coder data types such as `int16_T`.

```
#include "rtwtypes.h"

typedef enum {
    Reference=0,
    Working
} Dataset_T;

typedef struct DataStruct_tag {
  int16_T   OFFSET; /* OFFSET */
  int16_T   GAIN;   /* GAIN */
} DataStruct_type;
```

The file also declares the global variables and the functions.

Purpose of Custom Code

The code is designed so that the source code of a control algorithm (whether generated or handwritten) can read data from either `ReferenceStruct` or `WorkingStruct`

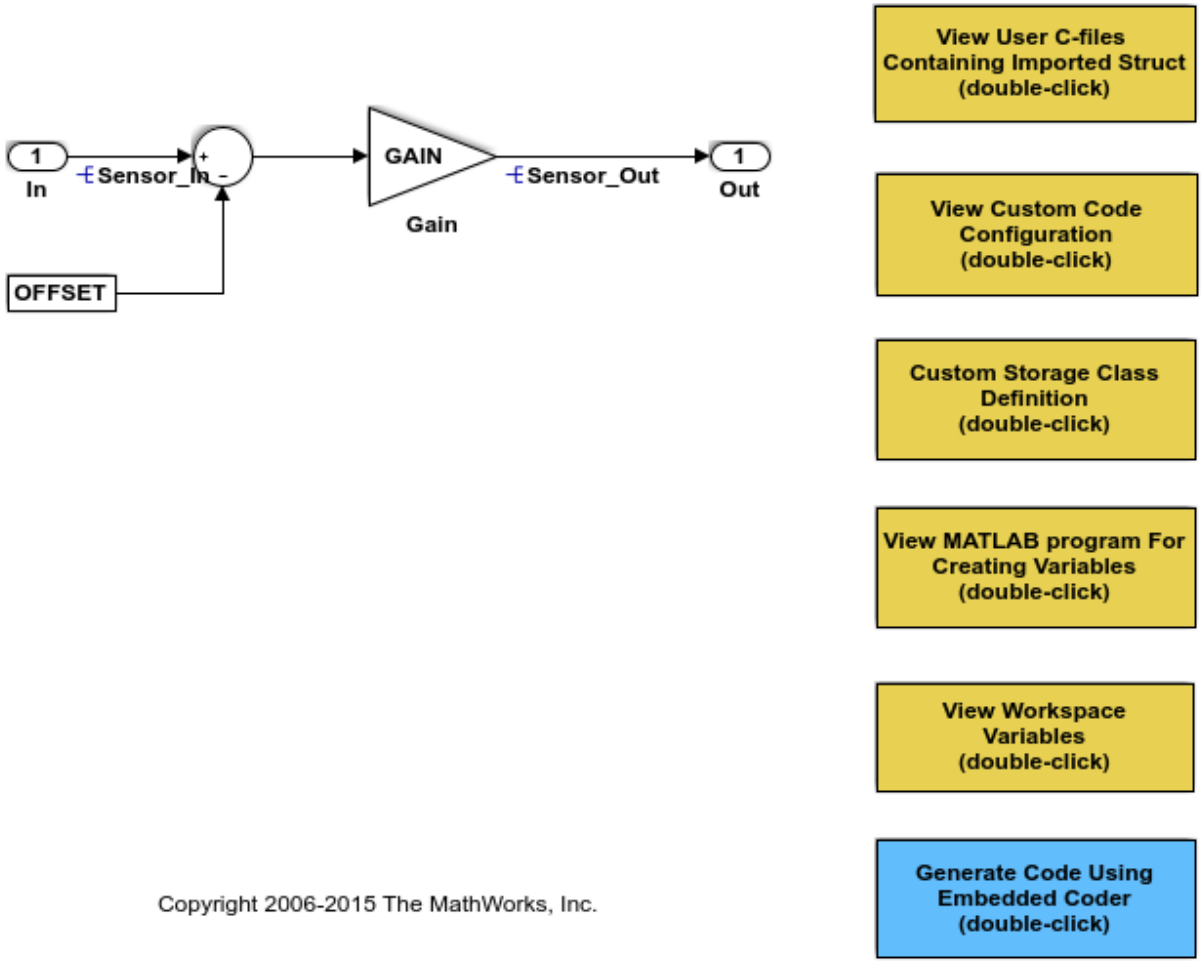
by dereferencing (->) `StructPointer`. A software engineer can write code that dynamically switches `StructPointer` between the address of `WorkingStruct` and `ReferenceStruct` by passing the corresponding enumeration member as the input argument in calls to the `SwitchStructPointer` function.

Later, in preparation for calibration while the algorithm executes, a calibration tool can make `StructPointer` point to `WorkingStruct`. The tool can then modify the fields of `WorkingStruct`.

If necessary for safety or in preparation for shutting down the application, the calibration tool can point `StructPointer` to `ReferenceStruct` instead. `ReferenceStruct` stores default parameter values that do not change during execution.

Explore Example Model

Open the example model, rtwdemo_importstruct.



The model creates variables and objects in the base workspace. The Constant block and the Gain block use the `ECoderDemos.Parameter` objects `GAIN` and `OFFSET` to set the **Constant value** and **Gain** block parameters. `ECoderDemos` is an example custom package that defines two classes, `Parameter` and `Signal`, and some custom storage classes.

In the Constant block dialog box, next to the value of the **Constant value** parameter, click the action button. Select **Open Variable**. In the property dialog box for `OFFSET`,

Storage class is set to `StructPointer`, which is a custom storage class that the `ECoderDemos` package defines. `GAIN` also uses this custom storage class.

Open the Custom Storage Class Designer and inspect the custom storage classes in the `ECoderDemos` package. At the command prompt, use this command:

```
cscdesigner('ECoderDemos')
```

This example package defines multiple custom storage classes, including `StructPointer`. You cannot edit the definitions. However, you can create your own packages and custom storage classes later. For an example that shows how to create a package and a custom storage class, see "Control Data Code by Creating Custom Storage Class".

Under **Custom storage class definitions**, click `StructPointer`. The settings for this custom storage class enable the generated code to interact with the pointer variable, `StructPointer`, from the custom code. For example, the custom storage class uses these settings:

- **Data scope** is set to `Imported` because the example custom code defines (allocates memory for) `StructPointer`. With this setting, the code generator avoids generating unnecessary, duplicate definitions for the data items, such as the `ECoderDemos.Parameter` objects, that use the custom storage class.
- **Data access** is set to `Pointer` because in the example custom code, `StructPointer` is a pointer.
- **Memory section** is set to `ConstVolatile` because the example custom code defines `StructPointer` as constant, volatile data (`const volatile`).
- **Type** is set to `FlatStructure` because in the example custom code, `StructPointer` points to a structure. With this setting, the generated code treats each data item (`ECoderDemos.Parameter` object) as a field of a flat structure whose variable name and type name you can specify.
- On the **Structure Attributes** tab, **Struct name** is set to `StructPointer`. For a `FlatStructure` custom storage class, **Struct name** specifies the name of the structure variable in the generated code. In this example, `StructPointer` is the name of the variable that the custom code defines.
- **Type name** is set to `DataStruct_type`, which is the name of the structure type that the example custom code defines.

In the model, in the **Configuration Parameters** dialog box, inspect the **Code Generation > Custom Code** pane.

Under **Insert custom C code in generated**, select **Initialize function**. In this model, this configuration parameter is set so that the generated code calls the `Init_WorkingStruct` function before execution of the primary algorithm. `Init_WorkingStruct` initializes the fields of `WorkingStruct` with the values from `ReferenceStruct`.

Under **Additional build information**, select **Source files**. This configuration parameter identifies the example custom code file `rtwdemo_importstruct_user.c` for inclusion in the build process after code generation.

Generate and Inspect Code

Generate code from the model.

In the generated file `rtwdemo_importstruct.c`, the model initialization function calls `Init_WorkingStruct`.

```
/* Model initialize function */
void rtwdemo_importstruct_initialize(void)
{
  /* user code (Initialize function Body) */

  /* Initialize the volatile memory data set before switching to it */
  Init_WorkingStruct();
}

/*
```

The algorithm in the model execution (`step`) function dereferences the pointer variable `StructPointer`.

```
/* Model step function */
void rtwdemo_importstruct_step(void)
{
  /* Gain: '<Root>/Gain' incorporates:
   *  Constant: '<Root>/Offset'
   *  Inport: '<Root>/In'
   *  Sum: '<Root>/Sum'
   */
  Sensor_Out = (int16_T)((int16_T)(Sensor_In - StructPointer->OFFSET) *
    StructPointer->GAIN);
```

```
}
```

Related Examples

- "Exchange Data Between External C/C++ Code and Simulink Model or Generated Code" on page 39-86
- "Create Tunable Calibration Parameter in the Generated Code" on page 19-60
- "Switch Between Sets of Parameter Values During Simulation and Code Execution" on page 19-103
- "Design Custom Storage Classes and Memory Sections" on page 23-34

Design Custom Storage Classes and Memory Sections

In this section...

Resources for Defining Custom Storage Classes

The resources for working with custom storage class definitions are:

- Use MATLAB class syntax to create a data class in a package. You can assign properties to the data class and add initialization code to enable custom storage class definition. For complete instructions, see "Define Data Classes" (Simulink).
- 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`, `Simulink.Parameter`, `Simulink.LookupTable`, and `Simulink.Breakpoint` classes; you do not need to use special object classes to generate code with CSCs.
- The Custom Storage Class Designer (`cscdesigner`) tool, which is described in this chapter. This tool lets you define CSCs that are tailored to your code generation requirements. The Custom Storage Class Designer provides a graphical user interface that you can use to implement CSCs. You can use your CSCs in code generation immediately, without a Target Language Compiler (TLC) or other programming. See "Design Custom Storage Classes and Memory Sections" on page 23-34 for details.

Create Packages for Custom Storage Class Definitions

Use MATLAB class syntax to create a data class in a package. You can assign properties to the data class and add initialization code to enable custom storage class definition. For complete instructions, see "Define Data Classes" (Simulink).

Use Custom Storage Class Designer

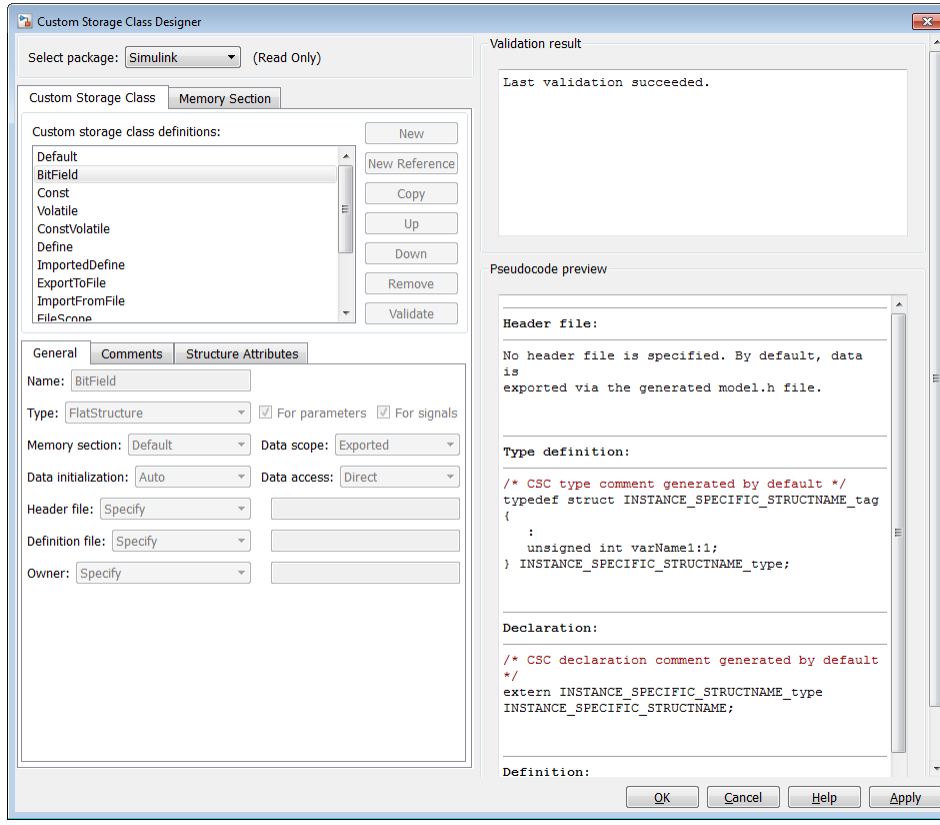
The Custom Storage Class Designer (`cscdesigner`) is a tool for creating and managing custom storage classes and memory sections. You can use the Custom Storage Class Designer to:

- 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
- Check a 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 for a particular package, type the following command at the MATLAB prompt:

```
cscdesigner ('mypkg')
```

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



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

- **Select package**: Lets you select from a menu of data class packages that have CSC definitions associated with them. See “Select Data Class Package” on page 23-37 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 “Manipulate Custom Storage Classes and Memory Sections” on page 23-38.

  - When the **Custom Storage Class** tab is selected, you can select a CSC definition or reference from a list and edit its properties. See "Edit Custom Storage Class Properties" on page 23-41 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 "Create and Edit Memory Section Definitions" on page 23-52 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 "Save Definitions" on page 23-40 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 "Preview Generated Code" on page 23-54 for details.
- **Validation result**: Displays errors encountered when the currently selected CSC definition is validated. See "Validate Definitions Category" on page 23-47 for details.

Select 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.

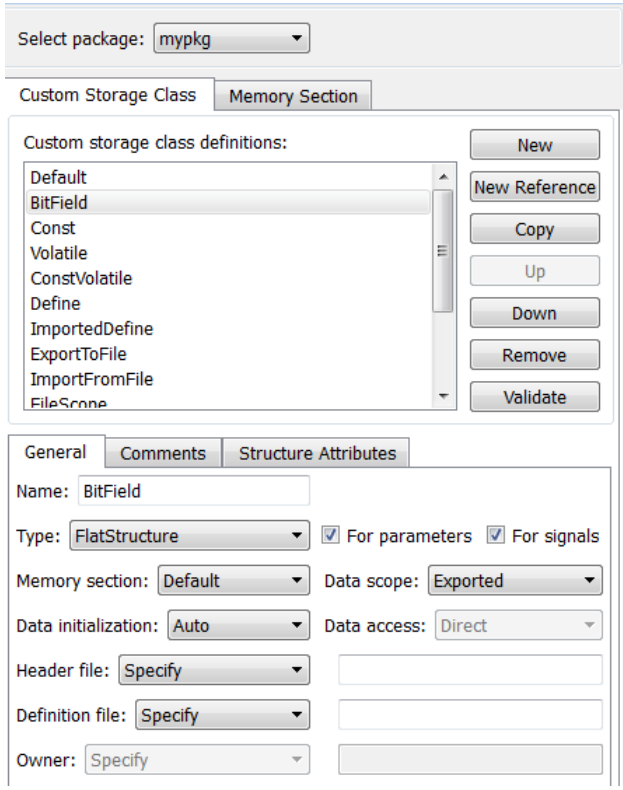
You need not search for or edit a CSC registration file directly: the Custom Storage Class Designer locates available CSC registration files. The **Select package** menu contains names of data class packages that have a CSC registration file on the MATLAB search path.

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

If you select a user-defined package, by default you can use the Custom Storage Class Designer to edit its custom storage classes and memory sections. If you select a built-in package, you cannot edit its custom storage classes or memory sections.

Manipulate Custom Storage Classes and Memory Sections

The **Custom Storage Class / Memory Section** panel lets you select, view, and (if the CSC is writable) edit CSC and memory section definitions and references. In the next figure and the subsequent examples, the selected package is `mypkg`. Instructions for creating a user-defined package like `mypkg` appear in "Design Custom Storage Classes and Memory Sections" on page 23-34.



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:

- "Edit Custom Storage Class Properties" on page 23-41 for information about the properties of the predefined CSCs.
- "Create and Edit Memory Section Definitions" on page 23-52 for information about the properties of the predefined memory section definitions.

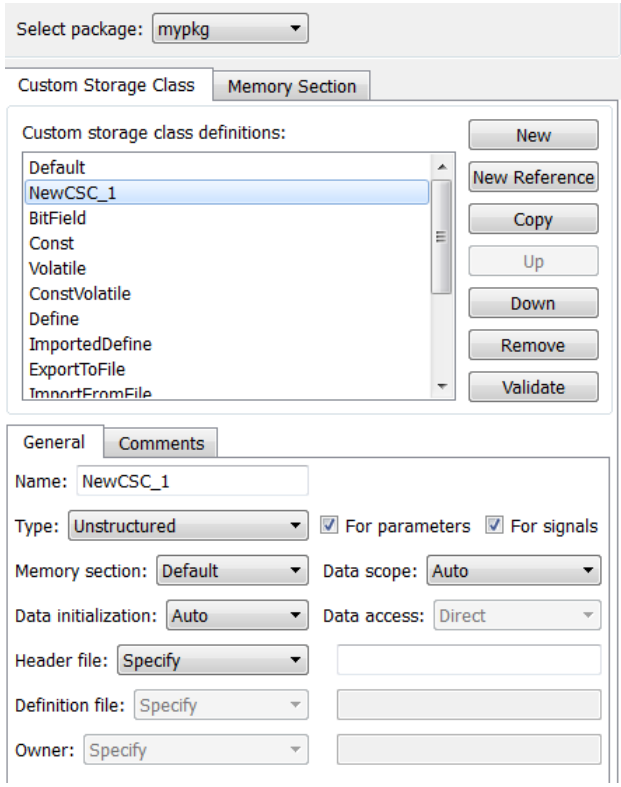
The buttons to the right of the list perform these functions, which are common to both custom storage classes and memory definitions:

- **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 "Use Custom Storage Class References" on page 23-47 and "Use Memory Section References" on page 23-55.
- **Copy**: Creates a copy of the selected definition / reference. The copy initially has a default name using the convention:

  ```
  definition_name_n
  ```

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

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



You can now rename the new class by typing the desired name into the **Name** field, and specify other fields.

Note: The class name must be a valid MATLAB variable name. See "Variable Names" (MATLAB)

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

Save Definitions

After you have created or edited a CSC or memory section definition or reference, you must save the changes to the CSC registration file. To do this, click **Save** in the

Filename panel. When you click **Save**, the current CSC and memory section definitions that are in memory are validated, and the definitions are written out.

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

Restart MATLAB After Changing Definitions

If you add, change, or delete custom storage class or memory section definitions for a user-defined class, and objects of that class already exist, you must restart MATLAB to use the changed definitions and to eliminate obsolete objects. When you save the changed definitions, a message appears indicating that you must restart MATLAB.

Edit Custom Storage Class Properties

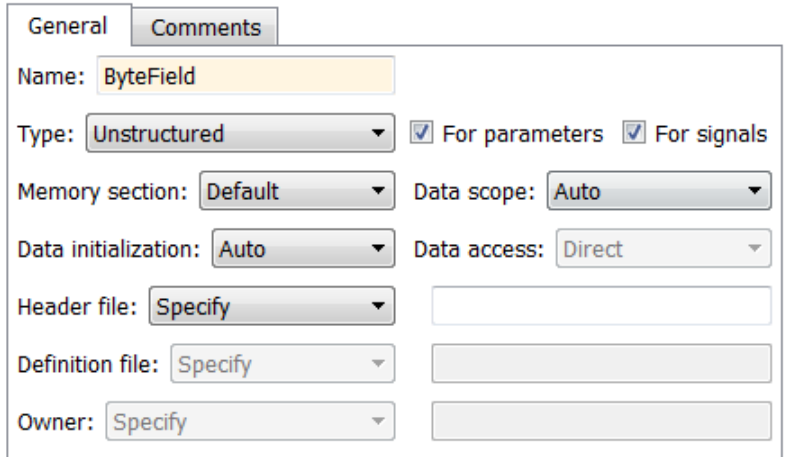
To view and edit the properties of a CSC, click the **Custom Storage Class** tab in the **Custom Storage Class / Memory Section** panel. Then, select a CSC name from the **Custom storage class definitions** list.

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 changes in 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 Category

Properties in the **General** category are common to CSCs. In the next figure and the subsequent examples, the selected custom storage class is `ByteField`. Instructions for creating a user-defined custom storage class like `ByteField` appear in “Manipulate Custom Storage Classes and Memory Sections” on page 23-38.



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

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

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

  - `FlatStructure`: Objects of this class are stored as members of a struct. A **Structure Attributes** tab is also displayed, allowing you to specify additional properties such as the struct name. See "Structure Attributes Category" on page 23-45.
  - `AccessFunction`: The generated code accesses objects of this class by calling custom `get` and `set` functions whose definitions you provide. See "Access Function Attributes Category" on page 23-46.
  - `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 TLC code. See "Define Advanced Custom Storage Classes Types" on page 23-78 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 "Create and Edit Memory Section Definitions" on page 23-52.
- **Data scope**: Controls the scope of symbols generated for data objects of this class. Values:
  - `Auto`: Symbol scope is determined internally by code generation. 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 a **Header File** is not 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`.
  - `File`: The scope of each symbol is the file that defines it. File scope requires each symbol to be used in a single file. If the same symbol is referenced in multiple files, an error occurs at code generation time.
  - `Instance specific`: Symbol scope is defined by the **Data scope** field of the `CoderInfo.CustomAttributes` property of each data object.
- **Data initialization**: Controls how storage is initialized in generated code. Values:
  - `Auto`: Storage initialization is determined internally by the code generation. Parameters have `Static` initialization, and signals have `Dynamic` initialization.
  - `None`: Initialization code is not 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 you set **Data scope** to `Imported`:

- To import a macro that you define by creating a preprocessor directive (`#define`), use the option **Header file** to configure the name of the header file that contains the directive.
- To import a macro that you define by configuring a compiler option, omit the header file.

  To specify the compiler option, use the model configuration parameter **Configuration Parameters** > **Code Generation** > **Custom Code** > **Additional build information** > **Defines**. See Code Generation Pane: Custom Code: Additional Build Information: Defines (Simulink Coder).

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

Note: The code generator might include dynamic initialization code for signals and states even if the CSC has **Data initialization** set to `None` or `Static`, if the initialization is required.

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

    ```
    extern myType myVariable;
    ```

  - `Pointer`: Symbols are declared as pointer variables, such as

    ```
    extern myType *myVariable;
    ```

  - `Instance specific`: Data access is defined by the **Data access** property of each data object.
- **Header file**: Defines the name of a header file that contains exported or imported variable declarations for objects of this class. If you set **Type** to `AccessFunction`, **Header file** defines the name of the header file that contains your `get` and `set` function prototypes. 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 not specify a 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 not specify a 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`, a `#include` directive for the header file is generated.

Comments Category

Comments

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

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

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

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

Structure Attributes Category

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.

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

Access Function Attributes Category

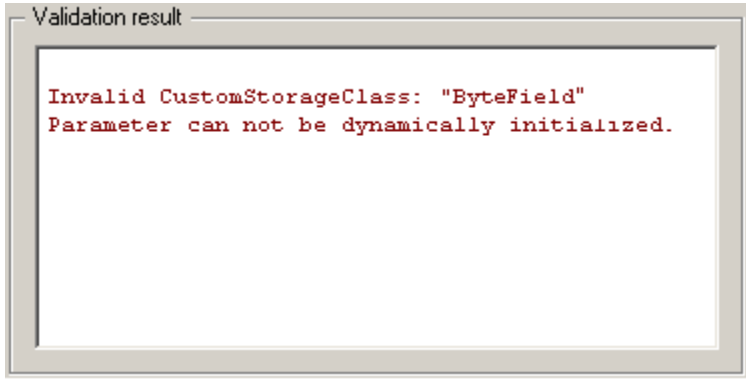
When you set **Type** to `AccessFunction` in the **General** panel, use the **Access Function Attributes** panel to control the `get` and `set` function names.

To apply the same `get` or `set` function naming scheme to all data items that use the custom storage class, set **Get function** or **Set function** to `Specify`. Then, in the new box, specify the function naming scheme, for example `get_myData_$N`. Use the token `$N` in each naming scheme to represent the name of each data item. If you do not use the token, each data item uses the same `get` or `set` function name, so the model generates an error when you generate code.

To specify a `get` or `set` function for each data item, set **Get function** or **Set function** to `Instance specific`. Later, when you create a data item and apply the custom storage class, specify the function name by configuring the custom attributes of the data item.

Validate Definitions Category

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



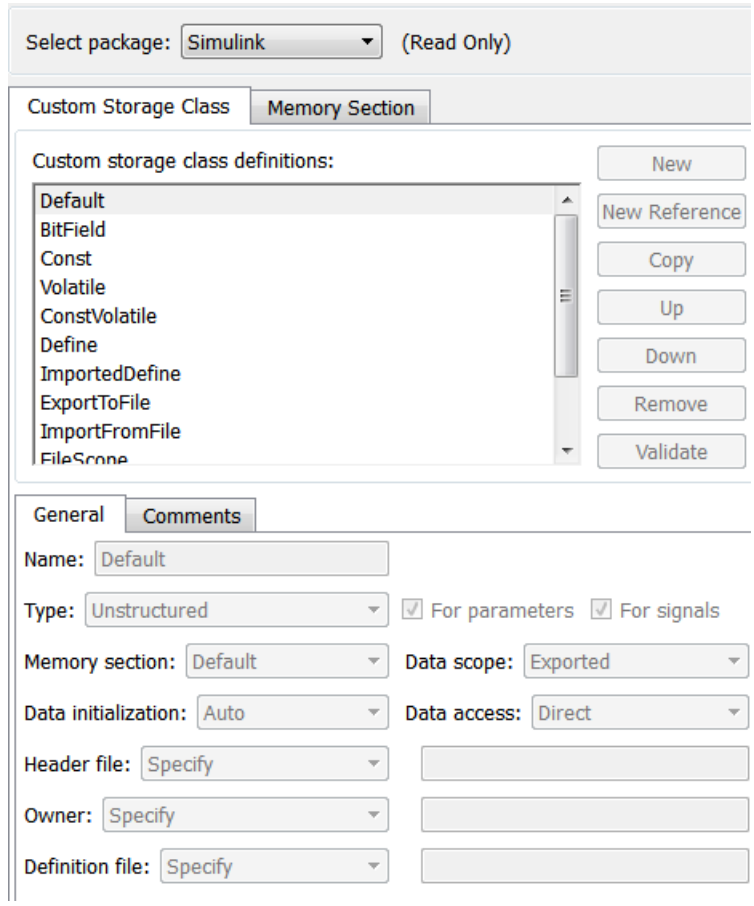
Validation is also performed whenever CSC definitions are saved. In this case, all CSC definitions are validated. (See “Save Definitions” on page 23-40.)

Use Custom Storage Class References

Packages can access and use custom storage classes that are defined in other packages, including both user-defined packages and predefined packages such as `Simulink`. Only one copy of the storage class exists, in the package that first defined it. Other packages refer to it by pointing to it in its original location. 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:

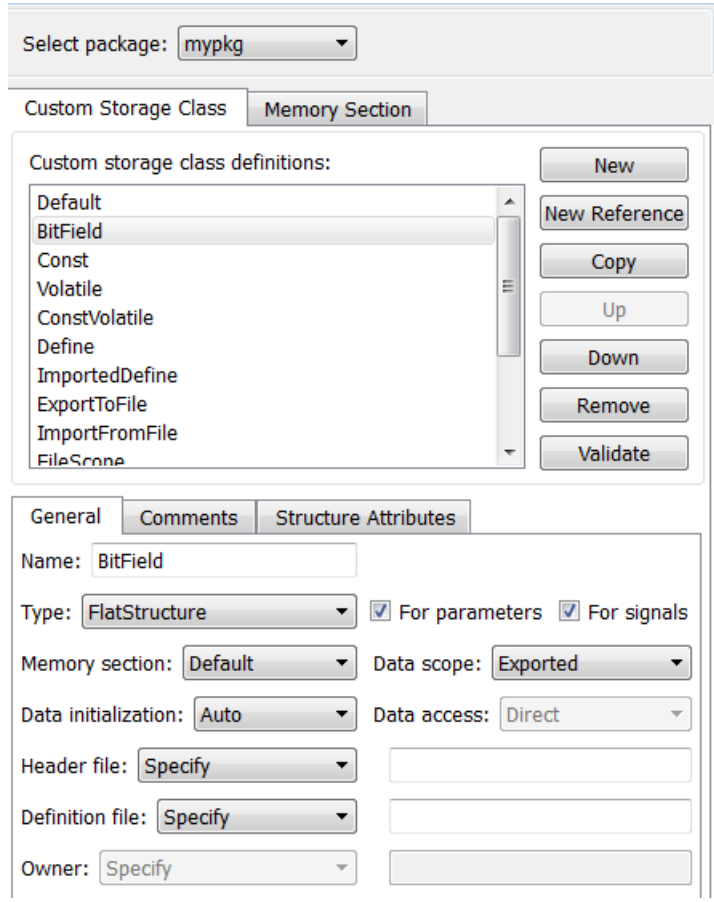
1. Type `cscdesigner` to launch the Custom Storage Class Designer.



2 Select the **Custom Storage Class** tab.

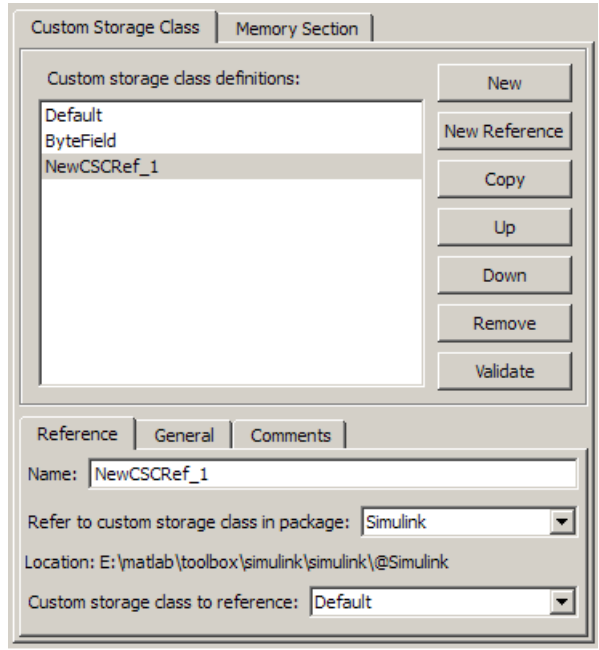
3 Use **Select Package** to select the package in which you want to reference a class or section defined in some other package. The selected package must be writable.

4 In the **Custom storage class definitions** pane, select the existing definition below which you want to insert the reference. For example:



5 Click **New Reference**.

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



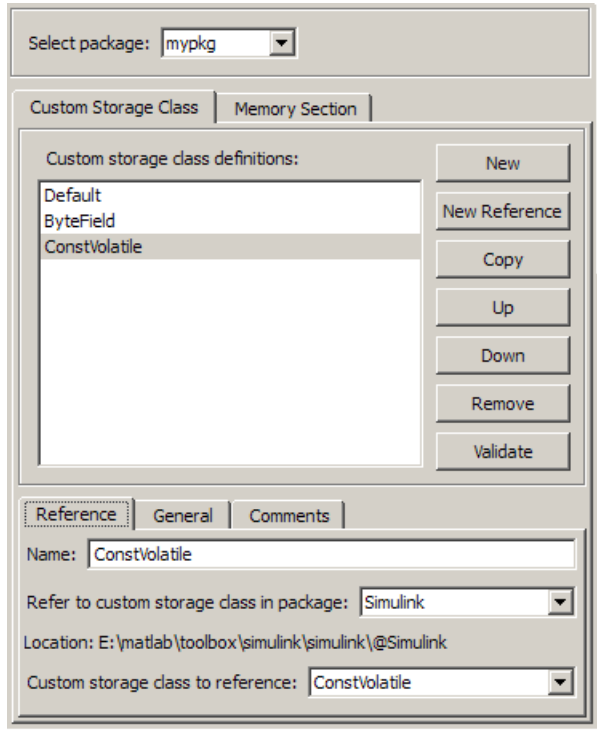
6 Use the **Name** field to enter a name for the new reference. The name must be unique in the importing package, but can duplicate the name in the source package. The name cannot be a TLC keyword. Violating this rule causes an error.

7 Set **Refer to custom storage class in package** to specify the package that contains the custom storage class you want to reference.

8 Set **Custom storage class to reference** to specify the custom storage class to be referenced. Trying to create a circular reference generates an error and leaves the package unchanged.

9 Click **OK** or **Apply** to save the changes to memory. See "Save Definitions" on page 23-40 for information about saving changes permanently.

For example, the next figure shows the custom storage class `ConstVolatile` imported from the `Simulink` package into `mypkg`, and given the same name that it has in the source package. Other names could have been used without affecting the properties of the storage class.



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

Change Existing Custom Storage Class 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 changes, then click **OK** or **Apply** to save the changes to memory. See "Save Definitions" on page 23-40 for information about saving changes permanently.

Protect Custom Storage Class Definitions

You can prevent changes to the custom storage class definitions of an entire data class package by converting the package CSC registration file from a MATLAB file to a P-file.

To learn more about CSC registration files, see “Custom Storage Class Implementation” on page 23-81.

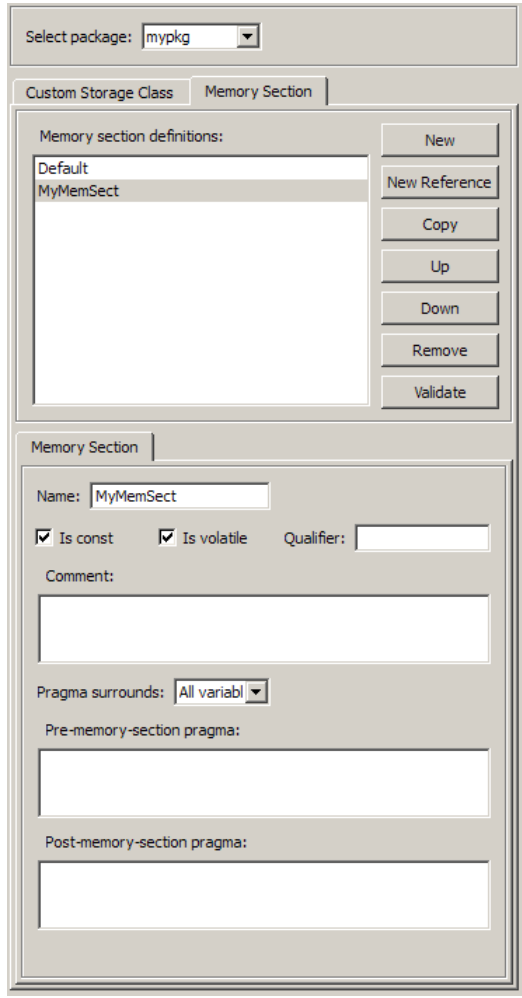
Create and Edit Memory Section Definitions

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

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

   A new memory section definition with a default name appears below the selected memory section.
5. Set the name and other properties of the memory section.
6. Click **OK** or **Apply**.

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



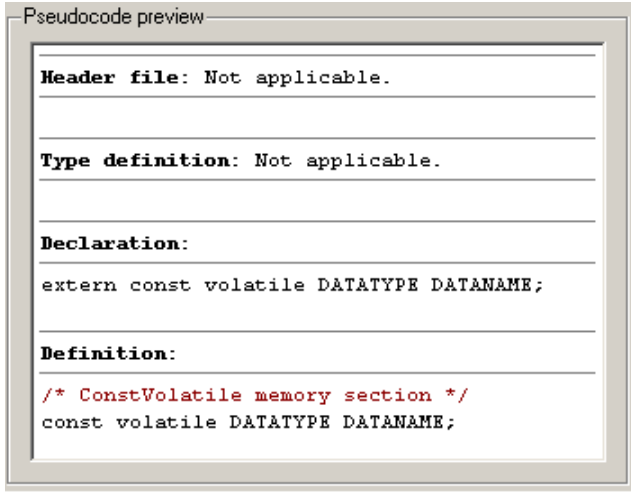
The **Memory section definitions** list lets you select a memory section definition to view or edit. The available memory section definitions also appear in the **Memory section name** menu in the **Custom Storage Class** panel. The properties of a memory section definition are as follows:

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

- **Is volatile**: If selected, a `volatile` qualifier is added to the symbol declarations.
- **Qualifier**: The text entered into this field is added to the symbol declarations as a further qualifier. Note that verification is not 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`.

Preview 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.



Use Memory Section References

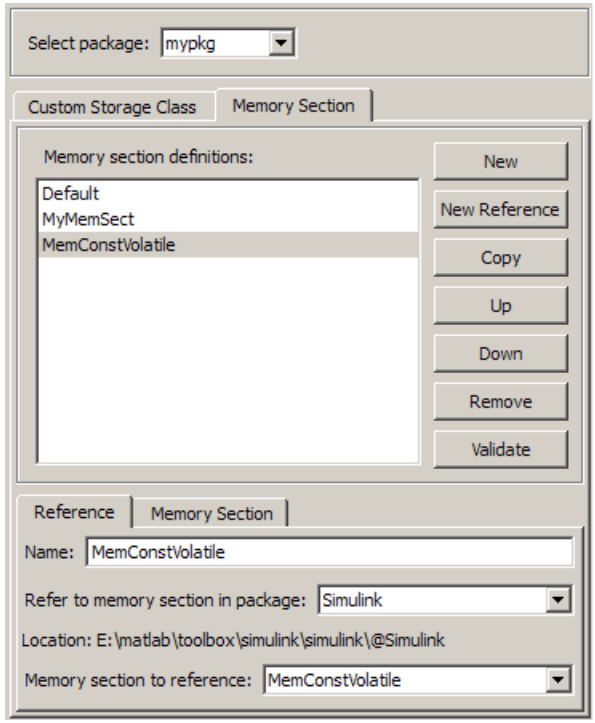
Packages can access and use memory sections that are defined in other packages, including both user-defined packages and predefined packages such as `Simulink`. Only one copy of the section exists, in the package that first defined it; other packages refer to it by pointing to it in its original location.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

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

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

   A new reference with a default name and properties appears below the previously selected definition. The new reference is selected, and a **Reference** tab appears that shows the reference's initial properties.
6. Use the **Name** field to enter a name for the new reference. The name must be unique in the importing package, but can duplicate the name in the source package.
7. Set **Refer to memory section in package** to specify the package that contains the memory section you want to reference.
8. Set **Memory section to reference** to specify the memory section to be referenced. Trying to create a circular reference generates an error and leaves the package unchanged.
9. Click **OK** or **Apply** to save the changes to memory. See “Save Definitions” on page 23-40 for information about saving changes permanently.

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



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

Change 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 changes, then click **OK** or **Apply** to save the changes to memory. See "Save Definitions" on page 23-40 for information about saving changes permanently.

Related Examples

- "Control Data Code by Creating Custom Storage Class" on page 23-73
- "Control Data Representation by Applying Custom Storage Classes" on page 23-58

- "Generate Code with Custom Storage Classes" on page 23-67
- "Data Objects" (Simulink)
- "Introduction to Custom Storage Classes" on page 23-2
- "Define Advanced Custom Storage Classes Types" on page 23-78
- "Access Structured Data Through a Pointer That External Code Defines" on page 23-27

Control Data Representation by Applying Custom Storage Classes

To control the declaration and definition of variables in the generated code, use the custom storage classes available with Embedded Coder. You can use custom storage classes to, for example, export multiple definitions or declarations to a single generated file, create structures and bit fields, and append storage type qualifiers to declarations.

To use custom storage classes, you can:

- Apply them to data objects, such as `Simulink.Parameter` and `Simulink.Signal`, that you associate with signal lines, block parameters, and states. See "Data Objects" (Simulink)

  You can also apply custom storage classes to `Simulink.LookupTable` and `Simulink.Breakpoint` objects, which you use to package lookup table data according to the ASAP2 and AUTOSAR standards.
- Specify them for signal lines and block states through dialog boxes and embedded signal objects. These techniques do not require you to store a data object in a workspace.

The custom storage classes then determine how the generated code represents the signals, parameters, and states.

To achieve a range of goals such as grouping variables into flat structures, or controlling declaration and definition file placement, use the custom storage classes from the built-in package `Simulink`. For more information about the capabilities of these custom storage classes, see "Simulink Package Custom Storage Classes" on page 23-5.

If the custom storage classes from the `Simulink` package do not satisfy your requirements, you can define your own custom storage classes. For basic information about defining your own custom storage class, see "Design Custom Storage Classes and Memory Sections" on page 23-34.

In this section...

In this section...

Apply a Custom Storage Class from the `Simulink` Package Using Data Objects

To apply a custom storage class from the built-in package `Simulink` to a signal, block parameter, or state:

1. Create a data object such as `Simulink.Parameter` or `Simulink.Signal`.
2. Configure the data object properties, including code generation settings. Specify the custom storage class.
3. Associate the data object with a signal, block parameter, or state in a model. For example:

   - In a block parameter dialog box, specify the name of a parameter data object.
   - Use the name of a signal data object to name a signal in a model. In the Signal Properties dialog box, select **Signal name must resolve to Simulink signal object**.

Specify Custom Storage Class for Data Object

1. In the Model Explorer **Model Hierarchy** pane, select the workspace that you want to contain the data object.
2. Click Add Parameter  to create a `Simulink.Parameter` object.
3. In the **Contents** pane, click the new data object, which is named `Param` by default.

4 In the **Dialog** pane, in the drop-down list **Storage class**, select `ExportToFile (Custom)`.

5 Under **Custom attributes**, specify additional code generation settings that the custom storage class requires. For example, specify **HeaderFile** as `myDataHdr.h`.

Programmatically Specify Custom Storage Class for Data Object

```
% Create a data object. For example, create a
% Simulink.Parameter object.
myParam = Simulink.Parameter(15.23);

% Specify the custom storage class called ExportToFile.
myParam.CoderInfo.StorageClass = 'Custom';
myParam.CoderInfo.CustomStorageClass = 'ExportToFile';

% Specify custom attributes for this data object.
myParam.CoderInfo.CustomAttributes.HeaderFile = 'myDataHdr.h';
```

Create and Apply Your Own Custom Storage Class Using Data Objects

To create your own custom storage class, you must create a data class package and define the custom storage class in the package. Afterward, you can apply the custom storage class to signals, block parameters, and states:

1 Create a data object from your data class package. For example, if you name your package `myPackage`, you create data objects such as `myPackage.Parameter` and `myPackage.Signal`.

2 Configure the data object properties, including code generation settings. Specify the custom storage class that you defined.

3 Associate the data object with a signal, block parameter, or state in a model. For example, specify the name of a parameter object in a block parameter dialog box, or use the name of a signal object to name a signal in a model.

For an example that shows how to control the generated code by creating and applying a custom storage class, see “Control Data Code by Creating Custom Storage Class” on page 23-73.

Specify Custom Storage Class for Data Object

Suppose that you define a data class package `myPackage` and a custom storage class `ExportDefToFile` in that package.

1. In the Model Explorer **Model Hierarchy** pane, select the workspace that you want to contain the data object.
2. Click the arrow next to Add Parameter  . In the drop-down list, select **Customize class lists**.
3. In the dialog box, under **Parameter classes**, select the check box next to `myPackage.Parameter`. Click **OK**.
4. Click the arrow next to Add Parameter again. In the drop-down list, select **myPackage Parameter**.

   A new data object appears in the workspace. The default object name is `Param`.
5. In the **Contents** pane, select the new data object. In the **Dialog** pane, in the drop-down list **Storage class**, select `ExportDefToFile (Custom)`.
6. Under **Custom attributes**, specify additional code generation settings that the custom storage class requires. For example, suppose that data objects that use `ExportDefToFile` require you to specify a definition file. You can specify **DefinitionFile** as `myDataSrc.c`.

Programmatically Specify Custom Storage Class for Data Object

Suppose that you define a data class package `myPackage` and a custom storage class `ExportDefToFile` in that package.

```
% Create a data object from your package. For example, create a
% myPackage.Parameter object.
myParam = myPackage.Parameter(15.23);

% Specify the custom storage class ExportDefToFile.
myParam.CoderInfo.StorageClass = 'Custom';
myParam.CoderInfo.CustomStorageClass = 'ExportDefToFile';

% Specify custom attributes for this data object. For example, suppose that
% ExportDefToFile requires a definition file for each data object.
myParam.CoderInfo.CustomAttributes.DefinitionFile = 'myDataSrc.c';
```

Apply Custom Storage Classes Directly to Signal Lines, Block States, and Outport Blocks

Through dialog boxes and the Model Data Editor (see “Configure Data Properties by Using a Table” (Simulink)), you can apply custom storage classes directly to signal lines

and block states. You do not need a data object that you store in a workspace or data dictionary. However, you cannot use a signal object in a workspace to specify other characteristics of the signal or state, such as data type.

To apply a storage class directly to a signal line, use the Signal Properties dialog box. For a block state, use the **State Attributes** tab in the block dialog box.

To apply a custom storage class from the built-in package `Simulink`:

1. Open the **Code Generation** tab in a Signal Properties dialog box, or the **State Attributes** tab in a block dialog box.
2. Specify a name in the **Signal name** box or the **State name** box. Click **Apply**.
3. In the **Storage class** drop-down list, select a custom storage class.

To apply a custom storage class from a different package:

1. Open the **Code Generation** tab in a Signal Properties dialog box or the **State Attributes** tab in a block dialog box.
2. Specify a name in the **Signal name** box or the **State name** box. Click **Apply**.
3. In the **Signal object class** drop-down list, choose a package by selecting a signal object class that the target package defines. For example, to apply custom storage classes from the built-in package `Simulink`, select `Simulink.Signal`.

   If the class that you want does not appear in the list:

   a. From the drop-down list, select `Customize class lists`.
   b. In the dialog box, under **Signal classes**, select the check box next to the class that you want. For example, to use custom storage classes from the built-in package `mpt`, select the check box next to `mpt.Signal`. Click **OK**.

      If you created your own package, the classes that the package defines appear in the dialog box only if you put the package folder in your current folder or on the MATLAB path.
   c. From the drop-down list, select the option that corresponds to what you selected. For example, select `mpt.Signal`.
4. In the **Storage class** drop-down list, select a custom storage class.

To apply a custom storage class to a root-level Outport block, use the Model Data Editor. You can also use the Model Data Editor to apply custom storage classes to signals

through a list that you can sort, group, and filter. See "Configure Data Interface by Using Model Data Editor" on page 23-69.

Programmatically Apply Custom Storage Classes Directly to Signals, States, and Outport Blocks Using Embedded Signal Objects

You can use *embedded signal objects* to apply custom storage classes to signal lines and block states. The embedded signal object does not appear in a workspace, so you do not need to save the object in a separate file. However, you can use embedded signal objects to specify only a custom storage class, the associated custom attributes, and an alias for the object. You must specify other signal or state characteristics, such as data type, in the source block dialog box.

If you create an embedded signal object for a signal or state, you cannot use a signal object in a workspace to specify other characteristics of the signal or state. The signal or state name resolves only to the embedded signal object.

To attach an embedded signal object to a signal or state:

1. Create a temporary signal object in a workspace such as the base workspace.
2. Specify a custom storage class and associated custom attributes.
3. Programmatically assign the object to:
   - The corresponding block outport if the target is a signal
   - The corresponding block state if the target is a state
4. Optionally, delete the temporary signal object from the workspace.

This example shows how to attach an embedded signal object to a signal in a model.

1. Open the example model `rtwdemo_secondOrderSystem`.

   ```
   rtwdemo_secondOrderSystem
   ```

2. Create a handle to the output of the block named Force: f(t).

   ```
   portHandles = get_param('rtwdemo_secondOrderSystem/Force: f(t)','PortHandles');
   outportHandle = portHandles.Outport;
   ```

3. Set the name of the corresponding signal to `ForceSignal`.

   ```
   set_param(outportHandle,'Name','ForceSignal')
   ```

4 In the base workspace, create a signal object and specify a custom storage class and relevant custom attributes.

```
tempObj = Simulink.Signal;
tempObj.CoderInfo.StorageClass = 'Custom';
tempObj.CoderInfo.CustomStorageClass = 'ExportToFile';
tempObj.CoderInfo.CustomAttributes.HeaderFile = 'myHdrFile.h';
```

You can create the object from the data class package `Simulink`, or from any other package, such as a package that you create.

5 Embed the signal object in the target signal line by attaching a copy of the temporary workspace object.

```
set_param(outportHandle,'SignalObject',tempObj);
```

6 Clear the object from the base workspace. The signal now uses an embedded copy of the object.

```
clear tempObj
```

To modify an existing embedded signal object, copy the object into the base workspace, modify the copy, and reattach the copy. For example, to change the custom storage class of the embedded signal object attached to the signal `ForceSignal`:

1 Copy the existing embedded signal object into the base workspace.

```
tempObj = get_param(outportHandle,'SignalObject');
```

2 Modify the properties of the object in the workspace.

```
tempObj.CoderInfo.CustomStorageClass='ImportFromFile';
tempObj.CoderInfo.CustomAttributes.HeaderFile = 'myOtherHdrFile.h';
```

3 Reattach a copy of the signal object.

```
set_param(outportHandle,'SignalObject',tempObj);
clear tempObj
```

To attach an embedded signal object to a root-level Outport block, using the function `set_param`, specify the block parameter `SignalName` to name the signal that the block represents. Use the parameter `SignalObject` to embed the signal object.

To attach an embedded signal object to a block state, using the `set_param` function, specify the block parameter `StateIdentifier` to name the state. Use the parameter `StateSignalObject` to embed the signal object.

To attach an embedded signal object to a data store that you define by using a Data Store Memory block, use the block parameter `StateSignalObject`. You do not need to specify a state name because the data store already has a name.

Specify Instance-Specific Attributes

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

Data objects have a property called `CoderInfo`, which stores an object of the class `Simulink.CoderInfo`. Instance-specific attributes are stored in the `Simulink.CoderInfo` property `CustomAttributes`. This property is initially defined as follows:

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

When you specify a custom storage class, Simulink automatically populates `CoderInfo.CustomAttributes` with fields to represent instance-specific attributes of that CSC. For example, if you set the storage class of a data object `MyObj` to `Struct`, then enter:

```
MyObj.CoderInfo.CustomAttributes
```

MATLAB displays:

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

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

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

MATLAB displays:

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

The table lists instance-specific attributes that the custom storage classes from the built-in package `Simulink` define. When a data object uses one of these custom storage classes, you can specify the corresponding instance-specific attribute values in the object.

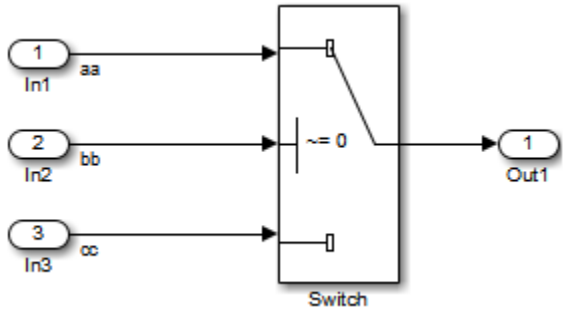
Custom Storage Class Name	Instance-Specific Attribute	Purpose
`BitField`	`CustomAttributes.StructName`	Name of the bitfield `struct` into which the code generator packs the object's Boolean data.
`ExportToFile`	`CustomAttributes.HeaderFile`	Name of header (`.h`) file that contains exported variable declarations and export directives for the object.
`GetSet`	`CustomAttributes.HeaderFile`	Name of header (`.h`) file to `#include` in the generated code. See "Access Data Through Functions with Custom Storage Class `GetSet`" on page 23-92.
	`CustomAttributes.GetFunction`	Specify the name of a function call to read data.
	`CustomAttributes.SetFunction`	Specify the name of a function call to write data.
`ImportedDefine`	`CustomAttributes.HeaderFile`	The header file that defines the values of code variant preprocessor conditionals. See "Generate Preprocessor Conditionals for Variant Systems" on page 14-33.
`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.

If you use a *grouped* custom storage class, you cannot specify many of its properties on an instance-specific basis. A grouped custom storage class combines multiple pieces of data into a single data structure. Data that use this format must have the same properties such as **Header file**, **Data scope**, and **Data initialization**. For example, the custom storage classes `BitField` and `Struct` represent multiple data objects in the generated code by using a single structure variable.

Generate Code with Custom Storage Classes

This example shows how to control data representation in the generate code by using custom storage classes and data objects.

Before you generate code for a model that uses custom storage classes, clear the **Configuration Parameters > All Parameters > Ignore custom storage classes** model option. Otherwise, the code generator ignores custom storage class specifications and treats data objects as if their **Storage Class** were `SimulinkGlobal`.

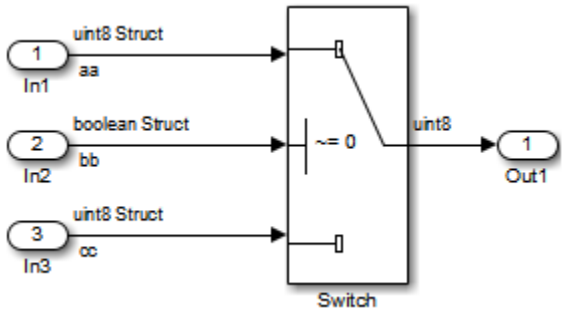


The model above contains three named signals: `aa`, `bb`, and `cc`. Using the `Struct` custom storage class, the example generates code that packs these signals into a `struct` named `mySignals`. The `struct` declaration is then exported to externally written code.

To specify the `struct`, you provide `Simulink.Signal` objects that specify the `Struct` custom storage class, and associate the objects with the signals as described in "Apply a Custom Storage Class from the `Simulink` Package Using Data Objects" on page 23-59. The three objects have the same properties. To view the properties, double-click one of the objects in the workspace browser (base workspace).

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

To display the storage class of the signals in the model, select **Display > Signals & Ports > Storage Class** in the Simulink editor. The figure below shows the block diagram with signal data types and signal storage classes displayed.



With the model's signal objects defined and associated with signals, you can generate code that uses the custom storage classes to generate the desired data structure for the signals. After code generation, the relevant definitions and declarations are located in three files:

- *model*`_types.h` defines the following `struct` type for storage of the three signals:

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

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

  ```
  /* 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;
```

Configure Data Interface by Using Model Data Editor

Use the Model Data Editor to apply storage classes to Inport and Outport blocks, signal lines, and Data Store Memory blocks. Use this technique to apply storage classes without locating the blocks and signals in the model and to configure the data interface of the model by using a single list.

To apply custom storage classes from a specific package, use the Model Explorer to create a signal object from the target package. Then, when you open the Model Data Editor, the **Storage class** column displays custom storage classes from the target package.

1. In the Model Explorer **Model Hierarchy** pane, select **Base Workspace**.
2. In the toolbar, click the arrow next to the **Add Signal**  button.
3. In the drop-down list, select `Customize class lists`.
4. In the **Customize class lists** dialog box, select a signal class from the target package. Click **OK**.
5. In the Model Explorer toolbar, click the arrow next to the **Add Signal** button.
6. In the drop-down list, select the target signal class.

   An object of the target signal class appears in the base workspace. Optionally, delete this unnecessary object.
7. Use the Model Data Editor to apply custom storage classes from the target package to other data items. In the Model Data Editor, in the **Storage class** column, the drop-down list allows you to select custom storage classes from the target package.

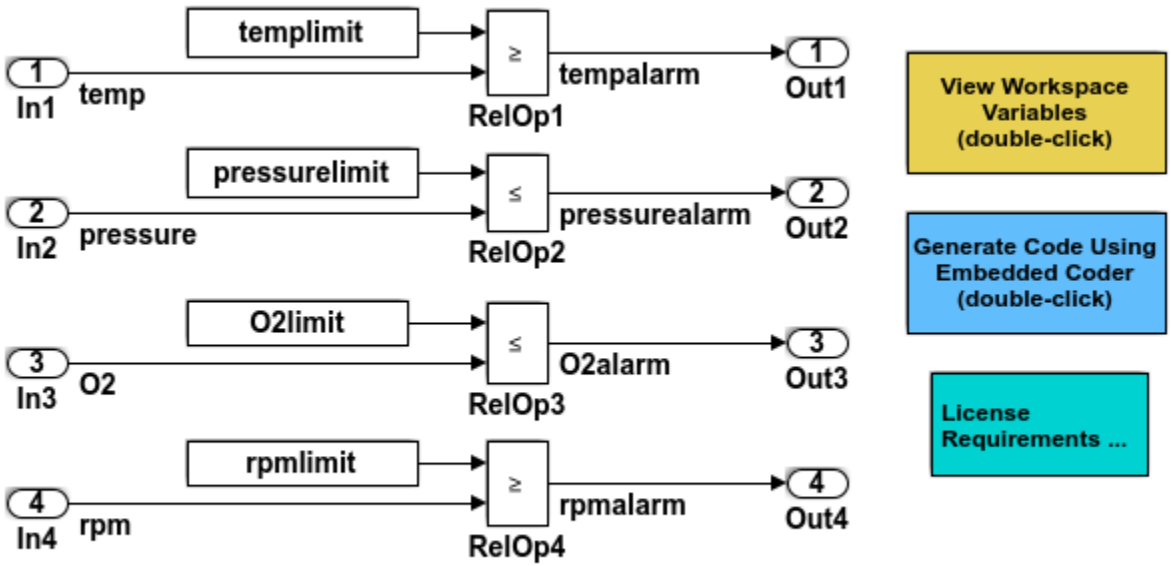
To learn how to use the Model Data Editor to configure a data interface, see “Use Model Data Editor to Configure Data Interface” on page 19-127.

Declare and Interface with Data Using Custom Storage Classes

Custom storage classes allow you to declare and interface with virtually any type of data. This model shows three of the several predefined custom storage classes provided with Embedded Coder. In this example, custom storage classes are specified for signals via the "Signal Properties..." dialog of a line and for parameters via Simulink parameter objects in the MATLAB Workspace.

Open the example model `rtwdemo_cscpredef`.

```
open_system('rtwdemo_cscpredef')
```



- The input signals use the custom storage class `Struct`.
- The constant parameters use the custom storage class `ConstVolatile`.
- The output signals use the custom storage class `BitField`.

You can use the Custom Storage Class Designer to:

- Create new custom storage classes

- Reference custom storage classes from other packages

To launch the Custom Storage Class Designer, type `cscdesigner` at the command prompt.

Specify Default `#include` Syntax for Data Header Files

To control the file placement of a data item such as a signal line or block state in the generated code, you can apply a custom storage class to the data item (see "Introduction to Custom Storage Classes" on page 23-2). You then use the `HeaderFile` custom attribute to specify the generated or custom header file that contains the declaration of the data.

To reduce maintenance effort and data entry, when you specify `HeaderFile`, you can omit delimiters (`"` or `<>`) and use only the file name. You can then control the default delimiters that the generated code uses for the corresponding `#include` directives. To use angle brackets by default, set **Configuration Parameters** > **Code Generation** > **Code Placement** > **#include file delimiters** to `#include <header.h>`.

Custom Storage Class Limitations

- Data objects cannot have a custom storage class and a multiword data type.
- The Fcn block does not support parameters with a custom storage class in code generation.
- For custom storage classes in models that use referenced models:
  - If you apply a grouped custom storage class to multiple data items, the custom storage class **Data scope** property must be set to `Imported` and you must provide the data declaration in a custom header file. Grouped custom storage classes use a single variable in the generated code to represent multiple data objects. For example, the custom storage classes `BitField` and `Struct` are grouped custom storage classes.
  - You cannot apply the custom storage class `FileScope` to parameters that referenced models use.
  - If data is assigned an ungrouped CSC, such as `Const`, and the data's **Data scope** property is `Exported`, its **Header file** property must be unspecified. This results in the data being exported with the standard header file, *`model`*`.h`. Note that for ungrouped data, the **Data scope** and **Header file** properties are either specified by the selected CSC, or as one of the data object's instance-specific properties.

- You cannot apply the custom storage class `FileScope` to data items used by a data exchange interface (C API, external mode, or ASAP2) or MAT-file logging. File-scoped data are not externally accessible.

Related Examples

- "Configure Data Interface by Applying Custom Storage Classes"
- "Control Data Code by Creating Custom Storage Class" on page 23-73
- "Exchange and Reuse Parameter Data Between Generated Code and Existing Code" on page 23-11
- "Design Custom Storage Classes and Memory Sections" on page 23-34
- "Data Objects" (Simulink)
- "Introduction to Custom Storage Classes" on page 23-2
- "Configure Generated Code According to Interface Control Document" on page 23-112

Control Data Code by Creating Custom Storage Class

When you integrate code generated from a model with existing code from another source, you can design custom storage classes to control the declaration and definition of model signals and block parameters. This example shows how to control code generated from a model by creating and applying your own custom storage class.

In this section...

Explore Example Model

Open the model rtwdemo_cscpredef. You can control code generated from this model by defining your own data classes and creating your own custom storage classes.

This example shows you how to export the declarations and definitions of multiple signals and parameters in the model to one declaration header file and one definition file.

Create Data Class Package

To create custom storage classes, you first create a data class package to contain the custom storage class definitions. Data objects created from your package can use all of the custom storage classes that the package defines.

1. Create your own data class package by copying the example package folder `+SimulinkDemos`. Navigate to the example package folder.

```
% Remember the current folder path
currentPath = pwd;

% Navigate to the example package folder
demoPath = '\toolbox\simulink\simdemos\dataclasses';
cd([matlabroot,demoPath])
```

2. Copy the `+SimulinkDemos` folder to your clipboard.

3. Return to your working folder.

   ```
   cd(currentPath)
   ```

4. Paste the `+SimulinkDemos` folder from your clipboard into your working folder. Rename the copied folder to `+myPackage`.
5. Navigate inside the `+myPackage` folder to the file `Signal.m` to edit the definition of the `Signal` class.
6. Uncomment the `methods` section that defines the method `setupCoderInfo`. In the call to the function `useLocalCustomStorageClasses`, replace `'packageName'` with `'myPackage'`. When you finish, the section appears as follows:

   ```
   methods
     function setupCoderInfo(h)
       % Use custom storage classes from this package
       useLocalCustomStorageClasses(h, 'myPackage');
     end
   end % methods
   ```

   The function `useLocalCustomStorageClasses` allows you to apply the custom storage classes that `myPackage` defines to data objects that you create from `myPackage`.
7. Save and close the file.
8. Navigate inside the `+myPackage` folder to the file `Parameter.m` to edit the definition of the `Parameter` class. Uncomment the `methods` section that defines the method `setupCoderInfo` and replace `'packageName'` with `'myPackage'`.
9. Save and close the file.

Create Custom Storage Class

You can use the Custom Storage Class Designer to create or to edit the custom storage classes that a data class package defines.

1. Set your current folder to the folder that contains the package `myPackage`.
2. Open the Custom Storage Class Designer.

   ```
   cscdesigner('myPackage')
   ```

3. Select the custom storage class `ExportToFile`.
4. In the **Name** field, rename the custom storage class to `ExportToGlobal`.

5 In the **Header file** drop-down list, change the selection from `Instance specific` to `Specify`. In the new field, provide the header file name `global.h`.

6 In the **Definition file** drop-down list, change the selection from `Instance specific` to `Specify`. In the new field, provide the definition file name `global.c`.

7 Click **OK**. Click **Yes** to save changes to the data class package `myPackage`.

Apply Custom Storage Class

To apply your own custom storage class, you create data objects from your package and configure the objects to use your custom storage class.

1 Create data objects to represent some of the parameters and signals in the example model. Create the objects using the data class package `myPackage`.

```
% Parameters
templimit = myPackage.Parameter(202);
pressurelimit = myPackage.Parameter(45.2);
O2limit = myPackage.Parameter(0.96);
rpmlimit = myPackage.Parameter(7400);

% Signals
tempalarm = myPackage.Signal;
pressurealarm = myPackage.Signal;
O2alarm = myPackage.Signal;
rpmalarm = myPackage.Signal;
```

2 Set the custom storage class of each object to `ExportToGlobal`.

```
% Parameters
templimit.CoderInfo.StorageClass = 'Custom';
templimit.CoderInfo.CustomStorageClass = 'ExportToGlobal';
pressurelimit.CoderInfo.StorageClass = 'Custom';
pressurelimit.CoderInfo.CustomStorageClass = 'ExportToGlobal';
O2limit.CoderInfo.StorageClass = 'Custom';
O2limit.CoderInfo.CustomStorageClass = 'ExportToGlobal';
rpmlimit.CoderInfo.StorageClass = 'Custom';
rpmlimit.CoderInfo.CustomStorageClass = 'ExportToGlobal';

% Signals
tempalarm.CoderInfo.StorageClass = 'Custom';
tempalarm.CoderInfo.CustomStorageClass = 'ExportToGlobal';
pressurealarm.CoderInfo.StorageClass = 'Custom';
pressurealarm.CoderInfo.CustomStorageClass = 'ExportToGlobal';
```

```
O2alarm.CoderInfo.StorageClass = 'Custom';
O2alarm.CoderInfo.CustomStorageClass = 'ExportToGlobal';
rpmalarm.CoderInfo.StorageClass = 'Custom';
rpmalarm.CoderInfo.CustomStorageClass = 'ExportToGlobal';
```

3 Select the **Signal name must resolve to Simulink signal object** option for each of the target signals in the model. You can select the option by using the Signal Properties dialog box or by using the command prompt.

```
% Signal tempalarm
portHandles = get_param('rtwdemo_cscpredef/RelOp1','PortHandles');
outputPortHandle = portHandles.Outport;
set_param(outputPortHandle,'MustResolveToSignalObject','on')

% Signal pressurealarm
portHandles = get_param('rtwdemo_cscpredef/RelOp2','PortHandles');
outputPortHandle = portHandles.Outport;
set_param(outputPortHandle,'MustResolveToSignalObject','on')

% Signal O2alarm
portHandles = get_param('rtwdemo_cscpredef/RelOp3','PortHandles');
outputPortHandle = portHandles.Outport;
set_param(outputPortHandle,'MustResolveToSignalObject','on')

% Signal rpmalarm
portHandles = get_param('rtwdemo_cscpredef/RelOp4','PortHandles');
outputPortHandle = portHandles.Outport;
set_param(outputPortHandle,'MustResolveToSignalObject','on')
```

Generate Code

1 Generate code for the example model.

```
rtwbuild('rtwdemo_cscpredef')
```

2 In the code generation report, view the generated header file `global.h`. The file contains the `extern` declarations of all of the model signals and parameters that use the custom storage class `ExportToGlobal`.

```
/* Declaration for custom storage class: ExportToGlobal */
extern boolean_T O2alarm;
extern real_T O2limit;
extern boolean_T pressurealarm;
extern real_T pressurelimit;
extern boolean_T rpmalarm;
```

```
extern real_T rpmlimit;
extern boolean_T tempalarm;
extern real_T templimit;
```

3 View the generated file `global.c`. The file contains the definitions of the model signals and parameters that use the custom storage class `ExportToGlobal`.

```
/* Definition for custom storage class: ExportToGlobal */
boolean_T O2alarm;
real_T O2limit = 0.96;
boolean_T pressurealarm;
real_T pressurelimit = 45.2;
boolean_T rpmalarm;
real_T rpmlimit = 7400.0;
boolean_T tempalarm;
real_T templimit = 202.0;
```

Related Examples

- "Generate Code with Custom Storage Classes" on page 23-67
- "Control Data Representation by Applying Custom Storage Classes" on page 23-58
- "Design Custom Storage Classes and Memory Sections" on page 23-34
- "Data Objects" (Simulink)
- "Introduction to Custom Storage Classes" on page 23-2
- "Define Advanced Custom Storage Classes Types" on page 23-78
- "Access Structured Data Through a Pointer That External Code Defines" on page 23-27

Define Advanced Custom Storage Classes Types

In this section...

Introduction

Certain data layouts, such as nested structures, cannot be generated using the standard `Unstructured` and `FlatStructure` custom storage class types. You can define an *advanced custom storage class* if you want to generate other types of data. Creating advanced CSCs requires understanding TLC programming and using a special advanced mode of the Custom Storage Class Designer. These sections explain how to define advanced CSC types. For more information about TLC programming, see "Why Use the Target Language Compiler?" (Simulink Coder).

For an example, see "Generate Code That Dereferences Data from a Literal Memory Address" on page 23-83.

Create Your Own Parameter and Signal Classes

The first step is to create your own package containing classes derived from `Simulink.Parameter` or `Simulink.Signal`. This procedure is described in "Define Data Classes" (Simulink).

Create Custom Attributes Classes for Custom Storage Classes

If you have instance-specific properties that are relevant only to your CSC, you should 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 instructions, see "Define Data Classes" (Simulink).

For example, the `ExportToFile` custom storage class requires that you set the `CoderInfo.CustomAttributes.HeaderFile` property to specify a `.h` file used for exporting each piece of data. See "Simulink Package Custom Storage Classes" on page 23-5 for further information on instance-specific properties.

Note: If you rename or remove custom attributes, you may need to manually edit the `csc_registration` file for the associated package to remove references to the custom attributes that you renamed or removed.

Write TLC Code for Custom Storage Classes

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

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

Register Custom Storage Class Definitions

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

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

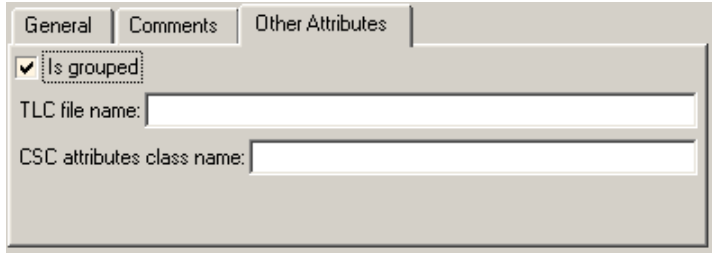
class types. The `Other` type cannot be assigned to a CSC except when the Custom Storage Class Designer is in advanced mode.

To register your class definitions:

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

   ```
   cscdesigner -advanced
   ```

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



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

4. Set the properties shown on the **Other Attributes** pane. The properties are:

   - **Is grouped**: Select this option if you intend to combine multiple data objects of this CSC into a single variable in the generated code. For example, the built-in custom storage classes `BitField` and `Struct` are grouped because they can represent multiple data objects in the generated code by using a single structure variable.
   - **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, for example, `myPackage.myCustomAttsClass` (see "Create Custom Attributes Classes for Custom Storage Classes" on page 23-78).

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).

Custom Storage Class Implementation

The file that defines a package's custom storage classes is called a *CSC registration file*. The file is named `csc_registration` and resides in the +*package* directory that defines the package. A CSC registration file can be a P-file (`csc_registration.p`) or a MATLAB file (`csc_registration.m`). A built-in package defines custom storage classes in both a P-file and a functionally equivalent MATLAB file. A user-defined package initially defines custom storage classes only in a MATLAB file.

P-files take precedence over MATLAB files, so when MATLAB looks for a package's CSC registration file and finds both a P-file and a MATLAB file, MATLAB loads the P-file and ignores the MATLAB file. The capabilities and tools, including the Custom Storage Class Designer, then use the CSC definitions stored in the P-file. P-files cannot be edited, so CSC Designer editing capabilities are disabled for CSCs stored in a P-file. If a P-file does not exist, MATLAB loads CSC definitions from the MATLAB file. MATLAB files are editable, so CSC Designer editing capabilities are enabled for CSCs stored in a MATLAB file.

Because CSC definitions for a built-in package exist in both a P-file and a MATLAB file, they are uneditable. You can make the definitions editable by deleting the P-file, but it is not recommended to modify built-in CSC registration files or other files under `matlabroot`. The preferred technique is to create packages, data classes, and custom storage classes, as described in “Define Data Classes” (Simulink).

The CSC Designer saves CSC definitions for user-defined packages in a MATLAB file, so the definitions are editable. You can make the definitions uneditable by using the `pcode` function to create an equivalent P-file, which will then shadow the MATLAB file. However, you should preserve the MATLAB file if you may need to make further changes, because you cannot modify CSC definitions that exist only in a P-file.

You can also use tools or techniques other than the Custom Storage Class Designer to create and edit MATLAB files that define CSCs. However, that practice is vulnerable to syntax errors and can give unexpected results. When MATLAB finds an older P-file that shadows a newer MATLAB file, it displays a warning in the MATLAB Command Window.

Related Examples

- “Introduction to the Target Language Compiler” (Simulink Coder)
- “Generate Code That Dereferences Data from a Literal Memory Address” on page 23-83
- “Control Data Code by Creating Custom Storage Class” on page 23-73
- “Control Data Representation by Applying Custom Storage Classes” on page 23-58
- “Generate Code with Custom Storage Classes” on page 23-67
- “Design Custom Storage Classes and Memory Sections” on page 23-34
- “Data Object Information in model.rtw” (Simulink Coder)

Generate Code That Dereferences Data from a Literal Memory Address

This example shows how to generate code that reads the value of a signal by dereferencing a memory address that you specify. With this technique, you can generate a control algorithm that interacts with memory that your hardware populates (for example, memory that stores the output of an analog-to-digital converter in a microcontroller).

In this example, you generate an algorithm that acquires input data from a 16-bit block of memory at address `0x8675309`. Assume that a hardware device asynchronously populates only the lower 10 bits of the address. The algorithm must treat the address as read-only (`const`), volatile (`volatile`) data, and ignore the upper 6 bits of the address.

The generated code can access the data by defining a macro that dereferences `0x8675309` and masks the unnecessary bits:

```
#define A2D_INPUT ((*(volatile const uint16_T *)0x8675309)&0x03FF)
```

To configure a model to generate code that defines and uses this macro, you must create an advanced custom storage class and write Target Language Compiler (TLC) code. For an example that shows how to use the Custom Storage Class Designer without writing TLC code, see “Control Data Code by Creating Custom Storage Class”.

Derivation of Macro Syntax

In this example, you configure the generated code to define and use the dereferencing macro. To determine the correct syntax for the macro, start by recording the target address.

```
0x8675309
```

Cast the address as a pointer to a 16-bit integer. Use the Simulink Coder data type name `uint16_T`.

```
(uint16_T *)0x8675309
```

Add the storage type qualifier `const` because the generated code must not write to the address. Add `volatile` because the hardware can populate the address at an arbitrary time.

```
(volatile const uint16_T *)0x8675309
```

Dereference the address.

```
*(volatile const uint16_T *)0x8675309
```

After the dereference operation, apply a mask to retain only the 10 bits that the hardware populates. Use explicit parentheses to control the order of operations.

```
(*(volatile const uint16_T *)0x8675309)&0x03FF
```

As a safe coding practice, wrap the entire construct in another layer of parentheses.

```
((*(volatile const uint16_T *)0x8675309)&0x03FF)
```

Create Example Model

Create the example model `ex_memmap_simple`.

For the Inport block, set the output data type to `uint16`. Name the signal as `A2D_INPUT`. The Inport block and the signal line represent the data that the hardware populates.

For the Gain block, set the output data type to `double`.

Create Package to Contain Definitions of Data Class and Custom Storage Class

In your current folder, create a folder named `+MemoryMap`. The folder defines a package named `MemoryMap`.

To make the package available for use outside of your current folder, you can add the `+MemoryMap` folder to the MATLAB path.

Create Custom Storage Class

To generate code that defines and reads `A2D_INPUT` as a macro, you must create a custom storage class that you can apply to the signal line in the model. Later, you write TLC code that complements the custom storage class.

Open the Custom Storage Class designer in advanced mode. To design a custom storage class that operates through custom TLC code, you must use the advanced mode.

```
cscdesigner('MemoryMap','-advanced');
```

In the Custom Storage Class Designer, click **New**. A new custom storage class, `NewCSC_1`, appears in the list of custom storage class definitions.

Rename the new custom storage class `MemoryMappedAddress`.

For `MemoryMappedAddress`, on the **General** tab, set:

- **Type** to `Other`. The custom storage class can operate through custom TLC code that you write later.
- **Data scope** to `Exported`. For data items that use this custom storage class, Simulink Coder generates the definition (for example, the `#define` statement that defines a macro).
- **Data initialization** to `None`. Simulink Coder does not generate code that initializes the data item. Use this setting because this custom storage class represents read-only data. You do not select `Macro` because the Custom Storage Class Designer does not allow you to use `Macro` for signal data.
- **Definition file** to `Specify` (leave the text box empty). For data items that consume memory in the generated code, **Definition file** specifies the `.c` source file that allocates the memory. However, this custom storage class yields a macro, which does not require memory. Header files (`.h`), not `.c` files, define macros. Setting **Definition file** to `Specify` instead of `Instance specific` prevents users of the custom storage class from unnecessarily specifying a definition file.
- **Header file** to `Instance specific`. To control the file placement of the macro definition, the user of the custom storage class must specify a header file for each data item that uses this custom storage class.
- **Owner** to `Specify` (leave the text box empty). **Owner** applies only to data items that consume memory.

After you finish selecting the settings, click **Apply** and **Save**.

Now, when you apply the custom storage class to a data item, such as the `A2D_INPUT` signal line, you can specify a header file to contain the generated macro definition. However, you cannot yet specify a memory address for the data item. To enable specification of a memory address, create a custom attributes class that you can associate with the `MemoryMappedAddress` custom storage class.

Define Class to Store Custom Attributes for Custom Storage Class

Define a MATLAB class to store additional information for data items that use the custom storage class. In this case, the additional information is the memory address.

In the `MemoryMap` package (the `+MemoryMap` folder), create a folder named `@MemoryMapAttribs`.

In the `@MemoryMapAttribs` folder, create a file named `MemoryMapAttribs`. The file defines a class that derives from the built-in class `Simulink.CustomStorageClassAttributes`.

```
classdef MemoryMapAttribs < Simulink.CustomStorageClassAttributes
    properties( PropertyType = 'char' )
        MemoryAddress = '';
    end
end
```

Later, you associate this MATLAB class with the `MemoryMappedAddress` custom storage class. Then, when you apply the custom storage class to a data item, you can specify a memory address.

Write TLC Code That Emits Correct C Code

Write TLC code that uses the attributes of the custom storage class, such as `HeaderFile` and `MemoryAddress`, to generate correct C code for each data item.

In the `+MemoryMap` folder, create a folder named `tlc`.

Navigate to the new folder.

Inspect the built-in template TLC file, TEMPLATE_v1.tlc.

```
edit(fullfile(matlabroot,...
    'toolbox','rtw','targets','ecoder','csc_templates','TEMPLATE_v1.tlc'))
```

Save a copy of `TEMPLATE_v1.tlc` in the `tlc` folder. Rename the copy `memory_map_csc.tlc`.

In `memory_map_csc.tlc`, find the portion that controls the generation of C-code data declarations.

```
    %case "declare"

      %% LibDefaultCustomStorageDeclare is the default declare function to
      %% declares a global variable whose identifier is the name of the data.
      %return "extern %<LibDefaultCustomStorageDeclare(record)>"
      %%break

    %% ============================================================================
```

The `declare` case (`%case`) constructs a return value (`%return`), which the code generator emits into the header file that you specify for each data item. To control the C code that declares each data item, adjust the return value in the `declare` case.

Replace the existing `%case` content with this new code, which specifies a different return value:

```
%case "declare"

        %% In TLC code, a 'record' is a data item (for example, a signal line).
        %% 'LibGetRecordIdentifier' returns the name of the data item.
        %assign id = LibGetRecordIdentifier(record)

        %assign dt = LibGetRecordCompositeDataTypeName(record)

        %% The 'CoderInfo' property of a data item stores a
        %% 'Simulink.CoderInfo' object, which stores code generation settings
        %% such as the storage class or custom storage class that you specify
        %% for the item.
        %assign ci = record.Object.ObjectProperties.CoderInfo
        %% The 'ci' variable now stores the 'Simulink.CoderInfo' object.

        %% By default, the 'CustomAttributes' property of a 'Simulink.CoderInfo'
        %% object stores a 'Simulink.CustomStorageClassAttributes' object.
        %% This nested object stores specialized code generation settings
        %% such as the header file and definition file that you specify for
        %% the data item.
        %%
        %% The 'MemoryMap' package derives a new class,
        %% 'MemoryMapAttribs', from 'Simulink.CustomStorageClassAttributes'.
        %% The new class adds a property named 'MemoryAddress'.
        %% This TLC code determines the memory address of the data item by
        %% acquiring the value of the 'MemoryAddress' property.
```

```
    %assign ca = ci.Object.ObjectProperties.CustomAttributes
    %assign address = ca.Object.ObjectProperties.MemoryAddress

    %assign width = LibGetDataWidth(record)

    %% This TLC code constructs the full macro, with correct C syntax,
    %% based on the values of TLC variables such as 'address' and 'dt'.
    %% This TLC code also asserts that the data item must be a scalar.
    %if width == 1
            %assign macro = ...
                "#define %<id> ((*(volatile const %<dt>*)%<address>) & 0x03FF)"
    %else
            %error( "Non scalars are not supported yet." )
    %endif

    %return "%<macro>"
  %%break

%% ============================================================================
```

The new TLC code uses built-in, documented TLC functions, such as `LibGetRecordIdentifier`, and other TLC commands and operations to access information about the data item. Temporary variables such as `dt` and `address` store that information. The TLC code constructs the full macro, with the correct C syntax, by expanding the variables, and stores the macro in the variable `macro`.

In the same file, find the portion that controls the generation of data definitions.

```
%case "define"

  %% LibDefaultCustomStorageDefine is the default define function to define
  %% a global variable whose identifier is the name of the data.  If the
  %% data is a parameter, the definition is also statically initialized to
  %% its nominal value (as set in MATLAB).
  %return "%<LibDefaultCustomStorageDefine(record)>"
  %%break

%% ============================================================================
```

The `define` case derives a return value that the code generator emits into a `.c` file, which defines data items that consume memory.

Replace the existing `%case` content with this new content:

```
%case "define"
  %return ""
  %%break

%% ============================================================================
```

`MemoryMappedAddress` yields a macro in the generated code, so you use the `declare` case instead of the `define` case to construct and emit the macro. To prevent the `define` case from emitting a duplicate macro definition, the new TLC code returns an empty string.

Complete the Definition of the Custom Storage Class

Your new MATLAB class, `MemoryMapAttribs`, can enable users of your new custom storage class, `MemoryMappedAddress`, to specify a memory address for each data item. To allow this specification, associate `MemoryMapAttribs` with `MemoryMappedAddress`. To generate correct C code based on the information that you specify for each data item, associate the customized TLC file, `memory_map_csc.tlc`, with `MemoryMappedAddress`.

Navigate to the folder that contains the `+MemoryMap` folder.

Open the Custom Storage Class Designer again.

For `MemoryMappedAddress`, on the **Other Attributes** tab, set:

- **TLC file name** to `memory_map_csc.tlc`.
- **CSC attributes class** to `MemoryMap.MemoryMapAttribs`.

Click **Apply** and **Save**.

Define Signal Data Class

To apply the custom storage class to a signal in a model, in the `MemoryMap` package, you must create a MATLAB class that derives from `Simulink.Signal`. When you configure the signal in the model, you select this new data class instead of the default class, `Simulink.Signal`.

In the `MemoryMap` package, create a folder named `@Signal`.

In the `@Signal` folder, create a file named `Signal.m`.

```
classdef Signal < Simulink.Signal
    methods
        function setupCoderInfo( this )
            useLocalCustomStorageClasses( this, 'MemoryMap' );
            return;
        end
    end
end
```

The file defines a class named `MemoryMap.Signal`. The class definition overrides the `setupCoderInfo` method, which the `Simulink.Signal` class already implements. The new implementation specifies that objects of the `MemoryMap.Signal` class use custom storage classes from the `MemoryMap` package (instead of custom storage classes from the `Simulink` package). When you configure a signal in a model by selecting the `MemoryMap.Signal` class, you can select the new custom storage class, `MemoryMappedAddress`.

Apply Custom Storage Class to Signal Line

Navigate to the folder that contains the example model and open the model.

In the model, select **View > Property Inspector**.

Click the signal named `A2D_INPUT`.

In the Property Inspector, under **Code Generation**, set **Signal object class** to `MemoryMap.Signal`. If you do not see `MemoryMap.Signal`, select `Customize class lists` and use the dialog box to enable the selection of `MemoryMap.Signal`.

In the Property Inspector, set **Storage class** to `MemoryMappedAddress`.

Set **Header file** to `memory_mapped_addresses.h`.

Set **MemoryAddress** to `0x8675309`.

Generate and Inspect Code

Generate code from the model.

```
### Starting build procedure for model: ex_memmap_simple
### Successful completion of build procedure for model: ex_memmap_simple
```

Inspect the generated header file `memory_mapped_addresses.h`. The file defines the macro `A2D_INPUT`, which corresponds to the signal line in the model.

```
/* Declaration of data with custom storage class MemoryMappedAddress */
#define A2D_INPUT                      ((*(volatile const uint16_T*)0x8675309) & 0x03F
```

Inspect the generated file `ex_memmap_simple.c`. The generated algorithmic code (which corresponds to the Gain block) calculates the model output, `rtY.Out1`, by operating on `A2D_INPUT`.

```
/* Model step function */
void ex_memmap_simple_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Gain: '<Root>/Gain'
   *  Inport: '<Root>/In1'
   */
  rtY.Out1 = 42.0 * (real_T)A2D_INPUT;
}
```

Related Examples

- "Signal Representation in Generated Code" on page 19-112
- "Access Data Through Functions with Custom Storage Class `GetSet`" on page 23-92
- "Choose an External Code Integration Workflow" on page 39-4
- "Define Advanced Custom Storage Classes Types" on page 23-78
- "Target Language Compiler"

Access Data Through Functions with Custom Storage Class `GetSet`

To integrate the generated code with legacy code that uses specialized functions to read from and write to data, you can use the custom storage class `GetSet`. Signals, block parameters, and states that use `GetSet` appear in the generated code as calls to accessor functions. You provide the function definitions.

To generate code that conforms to the AUTOSAR standard by accessing data through `Rte` function calls, use the Configure AUTOSAR Interface dialog box. See "AUTOSAR Interface Configuration".

Access Legacy Data Using `Get` and `Set` Functions

This example shows how to generate code that interfaces with legacy code by using specialized `get` and `set` functions to access data.

View the example legacy header file ComponentDataHdr.h. The file defines a large structure type `ComponentData`.

```
rtwdemodbtype('ComponentDataHdr.h','/* ComponentData */','} ComponentData;',1,1)
```

```
/* ComponentData */

typedef struct {
    ScalarData scalars;
    VectorData vectors;
    StructData structs;
    MatricesData matrices;
} ComponentData;
```

The field `scalars` is a substructure that uses the structure type `ScalarData`. The structure type `ScalarData` defines three scalar fields: `inSig`, `scalarParam`, and `outSig`.

```
rtwdemodbtype('ComponentDataHdr.h','/* ScalarData */','} ScalarData;',1,1)
```

```
/* ScalarData */
```

```
typedef struct {
    double inSig;
    double scalarParam;
    double outSig;
} ScalarData;
```

View the example legacy source file getsetSrc.c. The file defines and initializes a global variable `ex_getset_data` that uses the structure type `ComponentData`. The initialization includes values for the substructure `scalars`.

```
rtwdemodbtype('getsetSrc.c','/* Field "scalars" */','/* End of "scalars" */',1,1)
```

```
    /* Field "scalars" */
    {
    3.9,

    12.3,

    0.0
    },
    /* End of "scalars" */
```

The file also defines functions that read from and write to the fields of the substructure `scalars`. The functions simplify data access by dereferencing the leaf fields of the global structure variable `ex_getset_data`.

```
rtwdemodbtype('getsetSrc.c',...
    '/* Scalar get() and set() functions */','/* End of scalar functions */',1,1)
```

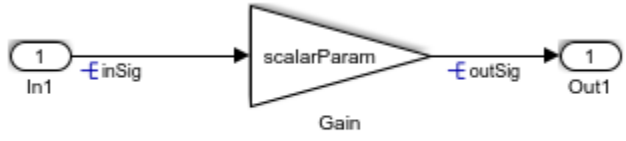
```
/* Scalar get() and set() functions */

double get_inSig(void)
{
    return ex_getset_data.scalars.inSig;
}

void set_inSig(double value)
{
    ex_getset_data.scalars.inSig = value;
}
```

```
double get_scalarParam(void)
{
    return ex_getset_data.scalars.scalarParam;
}

void set_scalarParam(double value)
{
    ex_getset_data.scalars.scalarParam = value;
}

double get_outSig(void)
{
    return ex_getset_data.scalars.outSig;
}

void set_outSig(double value)
{
    ex_getset_data.scalars.outSig = value;
}
```

View the example legacy header file getsetHdrScalar.h. The file contains the `extern` prototypes for the `get` and `set` functions defined in `getsetSrc.c`.

Open the example model rtwdemo_getset_scalar. The model creates the data objects `inSig`, `outSig`, and `scalarParam` in the base workspace. The objects correspond to the signals and parameter in the model.

```
open_system('rtwdemo_getset_scalar')
```



In the base workspace, double-click the object `inSig` to view its properties. The object uses the custom storage class `GetSet`. The `GetFunction` and `SetFunction` properties are set to the defaults, `get_$N` and `set_$N`. The generated code uses the function names that you specify in `GetFunction` and `SetFunction` to read from and write to the data.

The code replaces the token $N with the name of the data object. For example, for the data object inSig, the generated code uses calls to the legacy functions get_inSig and set_inSig.

For the data object inSig, the HeaderFile property is set to getsetHdrScalar.h. This legacy header file contains the get and set function prototypes. The data objects outSig and scalarParam also use the custom storage class GetSet and the header file getsetHdrScalar.h.

In the model Configuration Parameters dialog box, on the **Code Generation > Custom Code** pane, under **Include list of additional**, select **Source files**. The **Source files** box identifies the source file getsetSrc.c for inclusion during the build process. This legacy source file contains the get and set function definitions and the definition of the global structure variable ex_getset_data.

Generate code with the example model.

```
rtwbuild('rtwdemo_getset_scalar');
```

```
### Starting build procedure for model: rtwdemo_getset_scalar
### Successful completion of build procedure for model: rtwdemo_getset_scalar
```

In the code generation report, view the file rtwdemo_getset_scalar.c. The model step function uses the legacy get and set functions to execute the algorithm. The generated code accesses the legacy signal and parameter data by calling the custom, handwritten get and set functions.

```
rtwdemodbtype(fullfile('rtwdemo_getset_scalar_ert_rtw','rtwdemo_getset_scalar.c'),...
    '/* Model step function */','}',1,1)
```

```
/* Model step function */
void rtwdemo_getset_scalar_step(void)
{
  /* Gain: '<Root>/Gain' incorporates:
   *  Inport: '<Root>/In1'
   */
  set_outSig(get_scalarParam() * get_inSig());
}
```

You can generate code that calls your custom get and set functions as long as the functions that you write accept and return the expected values. For scalar data, the functions must have these characteristics:

- The `get` function must return a single scalar numeric value of the appropriate data type, and must not accept any arguments (`void`).
- The `set` function must not return anything (`void`), and must accept a single scalar numeric value of the appropriate data type.

Use `GetSet` with Vector Data

This example shows how to apply the custom storage class `GetSet` to signals and parameters that are vectors.

View the example legacy header file ComponentDataHdr.h. The file defines a large structure type `ComponentData`.

```
rtwdemodbtype('ComponentDataHdr.h','/* ComponentData */','} ComponentData;',1,1)
```

```
/* ComponentData */

typedef struct {
    ScalarData scalars;
    VectorData vectors;
    StructData structs;
    MatricesData matrices;
} ComponentData;
```

The field `vectors` is a substructure that uses the structure type `VectorData`. The structure type `VectorData` defines three vector fields: `inVector`, `vectorParam`, and `outVector`. The vectors each have five elements.

```
rtwdemodbtype('ComponentDataHdr.h','/* VectorData */','} VectorData;',1,1)
```

```
/* VectorData */

typedef struct {
    double inVector[5];
    double vectorParam[5];
    double outVector[5];
} VectorData;
```

View the example legacy source file getsetSrc.c. The file defines and initializes a global variable `ex_getset_data` that uses the structure type `ComponentData`. The initialization includes values for the substructure `vectors`.

```
rtwdemodbtype('getsetSrc.c','/* Field "vectors" */','/* End of "vectors" */',1,1)
```

```
    /* Field "vectors" */
    {
        {5.7, 6.8, 1.2, 3.5, 10.1},

        {12.3, 18.7, 21.2, 28, 32.9},

        {0.0, 0.0, 0.0, 0.0, 0.0}
    },
    /* End of "vectors" */
```

The file also defines functions that read from and write to the fields of the substructure `vectors`. The functions simplify data access by dereferencing the leaf fields of the global structure variable `ex_getset_data`. To access the vector data, all of the functions accept an integer index argument. The `get` function returns the vector value at the input index. The `set` function assigns the input `value` to the input index.

```
rtwdemodbtype('getsetSrc.c',...
    '/* Vector get() and set() functions */','/* End of vector functions */',1,1)
```

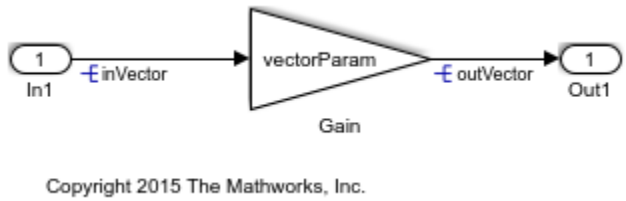
```
/* Vector get() and set() functions */

double get_inVector(int index)
{
    return ex_getset_data.vectors.inVector[index];
}

void set_inVector(int index, double value)
{
    ex_getset_data.vectors.inVector[index] = value;
}

double get_vectorParam(int index)
{
    return ex_getset_data.vectors.vectorParam[index];
}

void set_vectorParam(int index, double value)
{
    ex_getset_data.vectors.vectorParam[index] = value;
}
```

```
double get_outVector(int index)
{
    return ex_getset_data.vectors.outVector[index];
}

void set_outVector(int index, double value)
{
       ex_getset_data.vectors.outVector[index] = value;
}
```

View the example legacy header file getsetHdrVector.h. The file contains the `extern` prototypes for the `get` and `set` functions defined in `getsetSrc.c`.

Open the example model rtwdemo_getset_vector. The model creates the data objects `inVector`, `outVector`, and `vectorParam` in the base workspace. The objects correspond to the signals and parameter in the model.

```
open_system('rtwdemo_getset_vector')
```



In the base workspace, double-click the object `inVector` to view its properties. The object uses the custom storage class `GetSet`. The property `HeaderFile` is specified as `getsetHdrVector.h`. This legacy header file contains the `get` and `set` function prototypes.

In the model Configuration Parameters dialog box, on the **Code Generation > Custom Code** pane, the example legacy source file `getsetSrc.c` is identified for inclusion during the build process. This legacy source file contains the `get` and `set` function definitions and the definition of the global structure variable `ex_getset_data`.

Generate code with the example model.

```
rtwbuild('rtwdemo_getset_vector');

### Starting build procedure for model: rtwdemo_getset_vector
### Successful completion of build procedure for model: rtwdemo_getset_vector
```

In the code generation report, view the file `rtwdemo_getset_vector.c`. The model `step` function uses the legacy `get` and `set` functions to execute the algorithm.

```
rtwdemodbtype(fullfile('rtwdemo_getset_vector_ert_rtw','rtwdemo_getset_vector.c'),...
    '/* Model step function */','}',1,1)


/* Model step function */
void rtwdemo_getset_vector_step(void)
{
  int32_T i;

  /* Gain: '<Root>/Gain' incorporates:
   *  Inport: '<Root>/In1'
   */
  for (i = 0; i < 5; i++) {
    set_outVector( i , get_vectorParam( i ) * get_inVector( i ));
  }
```

When you use the custom storage class `GetSet` with vector data, the `get` and `set` functions that you provide must accept an index input. The `get` function must return a single element of the vector. The `set` function must write to a single element of the vector.

Use `GetSet` with Structured Data

This example shows how to apply the custom storage class `GetSet` to nonvirtual bus signals and structure parameters in a model.

View the example legacy header file ComponentDataHdr.h. The file defines a large structure type `ComponentData`.

```
rtwdemodbtype('ComponentDataHdr.h','/* ComponentData */','} ComponentData;',1,1)


/* ComponentData */
```

```
typedef struct {
    ScalarData scalars;
    VectorData vectors;
    StructData structs;
    MatricesData matrices;
} ComponentData;
```

The field `structs` is a substructure that uses the structure type `StructData`. The structure type `StructData` defines three fields: `inStruct`, `structParam`, and `outStruct`.

```
rtwdemodbtype('ComponentDataHdr.h','/* StructData */','} StructData;',1,1)
```

```
/* StructData */

typedef struct {
    SigBus inStruct;
    ParamBus structParam;
    SigBus outStruct;
} StructData;
```

The fields `inStruct`, `structParam`, and `outStruct` are also substructures that use the structure types `SigBus` and `ParamBus`. Each of these two structure types define three scalar fields.

```
rtwdemodbtype('ComponentDataHdr.h','/* SigBus */','} ParamBus',1,1)
```

```
/* SigBus */

typedef struct {
    double cmd;
    double sensor1;
    double sensor2;
} SigBus;

/* ParamBus */

typedef struct {
    double offset;
    double gain1;
    double gain2;
} ParamBus;
```

View the example legacy source file getsetSrc.c. The file defines and initializes a global variable `ex_getset_data` that uses the structure type `ComponentData`. The initialization includes values for the substructure `structs`.

```
rtwdemodbtype('getsetSrc.c','/* Field "structs" */','/* End of "structs" */',1,1)
```

```
    /* Field "structs" */
    {
        {1.3, 5.7, 9.2},

        {12.3, 9.6, 1.76},

        {0.0, 0.0, 0.0}
    },
    /* End of "structs" */
```

The file also defines functions that read from and write to the fields of the substructure `structs`. The functions simplify data access by dereferencing the fields of the global structure variable `ex_getset_data`. The functions access the data in the fields `inStruct`, `structParam`, and `outStruct` by accepting and returning complete structures of the types `SigBus` and `ParamBus`.

```
rtwdemodbtype('getsetSrc.c',...
    '/* Structure get() and set() functions */','/* End of structure functions */',1,1)
```

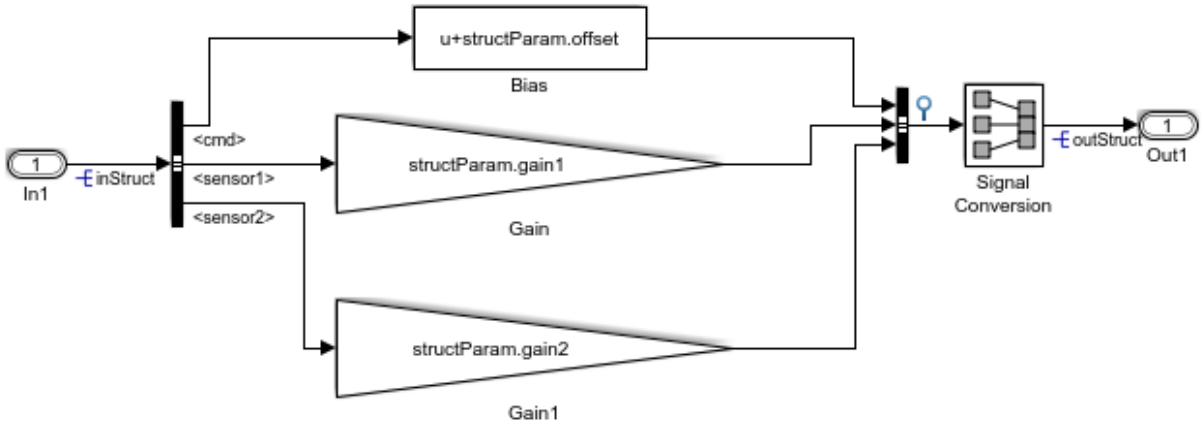
```
/* Structure get() and set() functions */

SigBus get_inStruct(void)
{
    return ex_getset_data.structs.inStruct;
}

void set_inStruct(SigBus value)
{
    ex_getset_data.structs.inStruct = value;
}

ParamBus get_structParam(void)
{
    return ex_getset_data.structs.structParam;
}
```

```
void set_structParam(ParamBus value)
{
    ex_getset_data.structs.structParam = value;
}

SigBus get_outStruct(void)
{
    return ex_getset_data.structs.outStruct;
}

void set_outStruct(SigBus value)
{
    ex_getset_data.structs.outStruct = value;
}
```

View the example legacy header file getsetHdrStruct.h. The file contains the `extern` prototypes for the `get` and `set` functions defined in `getsetSrc.c`.

Open the example model rtwdemo_getset_struct. The model creates the data objects `inStruct`, `structParam`, and `outStruct` in the base workspace. The objects correspond to the signals and parameter in the model.

```
open_system('rtwdemo_getset_struct')
```



In the base workspace, double-click the object `inStruct` to view its properties. The object uses the custom storage class `GetSet`. The property `HeaderFile` is specified as `getsetHdrStruct.h`. This legacy header file contains the `get` and `set` function prototypes.

The model also creates the bus objects `ParamBus` and `SigBus` in the base workspace. The signals and parameter in the model use the bus types that these objects define. The property `DataScope` of each bus object is set to `Imported`. The property `HeaderFile` is set to `ComponentDataHdr.h`. The generated code imports these structure types from the legacy header file `ComponentDataHdr.h`.

In the model Configuration Parameters dialog box, on the **Code Generation > Custom Code** pane, the example legacy source file `getsetSrc.c` is identified for inclusion during the build process. This legacy source file contains the `get` and `set` function definitions and the definition of the global structure variable `ex_getset_data`.

Generate code with the example model.

```
rtwbuild('rtwdemo_getset_struct');
```

```
### Starting build procedure for model: rtwdemo_getset_struct
### Successful completion of build procedure for model: rtwdemo_getset_struct
```

In the code generation report, view the file `rtwdemo_getset_struct.c`. The model `step` function uses the legacy `get` and `set` functions to execute the algorithm.

```
rtwdemodbtype(fullfile('rtwdemo_getset_struct_ert_rtw','rtwdemo_getset_struct.c'),...
    '/* Model step function */','}',1,1)
```

```
/* Model step function */
void rtwdemo_getset_struct_step(void)
{
  /* Bias: '<Root>/Bias' incorporates:
   *  Inport: '<Root>/In1'
   */
  rtDW.BusCreator.cmd = get_inStruct().cmd + get_structParam().offset;

  /* Gain: '<Root>/Gain' incorporates:
   *  Inport: '<Root>/In1'
   */
  rtDW.BusCreator.sensor1 = get_structParam().gain1 * get_inStruct().sensor1;
```

```
  /* Gain: '<Root>/Gain1' incorporates:
   *  Inport: '<Root>/In1'
   */
  rtDW.BusCreator.sensor2 = get_structParam().gain2 * get_inStruct().sensor2;

  /* SignalConversion: '<Root>/Signal Conversion' */
  set_outStruct(rtDW.BusCreator);
}
```

When you use the custom storage class `GetSet` with structured data, the `get` and `set` functions that you provide must return and accept complete structures. The generated code dereferences individual fields of the structure that the `get` function returns.

The output signal of the Bus Creator block is a test point. This signal is the input for a Signal Conversion block. The test point and the Signal Conversion block exist so that the generated code defines a variable for the output of the Bus Creator block. To provide a complete structure argument for the function `set_outStruct`, you must configure the model to create this variable.

Use `GetSet` with Matrix Data

This example shows how to apply the custom storage class `GetSet` to signals and parameters that are matrices.

View the example legacy header file ComponentDataHdr.h. The file defines a large structure type `ComponentData`.

```
rtwdemodbtype('ComponentDataHdr.h',...
    '/* ComponentData */','} ComponentData;',1,1)
```

```
/* ComponentData */

typedef struct {
    ScalarData scalars;
    VectorData vectors;
    StructData structs;
    MatricesData matrices;
} ComponentData;
```

The field `matrices` is a substructure that uses the structure type `MatricesData`. The structure type `MatricesData` defines three fields: `matrixInput`, `matrixParam`, and

`matrixOutput`. The fields store matrix data as serial arrays. In this case, the input and parameter fields each have 15 elements. The output field has nine elements.

```
rtwdemodbtype('ComponentDataHdr.h'...
    ,'/* MatricesData */','} MatricesData;',1,1)
```

```
/* MatricesData */

typedef struct {
    double matrixInput[15];
    double matrixParam[15];
    double matrixOutput[9];
} MatricesData;
```

View the example legacy source file getsetSrc.c. The file defines and initializes a global variable `ex_getset_data` that uses the structure type `ComponentData`. The initialization includes values for the substructure `matrices`.

```
rtwdemodbtype('getsetSrc.c',...
    '/* Field "matrices" */','/* End of "matrices" */',1,1)
```

```
    /* Field "matrices" */
    {
        {12.0, 13.9, 7.4,
         0.5, 11.8, 6.4,
         4.7, 5.3, 13.0,
         0.7, 16.1, 13.5,
         1.6, 0.5, 3.1},

        {8.3, 12.0, 11.5, 2.0, 5.7,
         7.5, 12.8, 11.1, 8.4, 9.9,
         10.9, 4.6, 2.7, 16.3, 3.8},

        {0.0, 0.0, 0.0,
         0.0, 0.0, 0.0,
         0.0, 0.0, 0.0}
    }
    /* End of "matrices" */
```

The input matrix has five rows and three columns. The matrix parameter has three rows and five columns. The matrix output has three rows and three columns. The file defines macros that indicate these dimensions.

```
rtwdemodbtype('getsetSrc.c',...
    '/* Matrix dimensions */','/* End of matrix dimensions */',1,1)


/* Matrix dimensions */

#define MATRIXINPUT_NROWS 5
#define MATRIXINPUT_NCOLS 3

#define MATRIXPARAM_NROWS 3
#define MATRIXPARAM_NCOLS 5

#define MATRIXOUTPUT_NROWS MATRIXPARAM_NROWS
#define MATRIXOUTPUT_NCOLS MATRIXINPUT_NCOLS
```

The file also defines functions that read from and write to the fields of the substructure `matrices`.

```
rtwdemodbtype('getsetSrc.c',...
    '/* Matrix get() and set() functions */','/* End of matrix functions */',1,1)


/* Matrix get() and set() functions */

double get_matrixInput(int colIndex)
{
    int rowIndexGetInput = MATRIXINPUT_NCOLS * (colIndex % MATRIXINPUT_NROWS) + colInde
    return ex_getset_data.matrices.matrixInput[rowIndexGetInput];
}

void set_matrixInput(int colIndex, double value)
{
    int rowIndexSetInput = MATRIXINPUT_NCOLS * (colIndex % MATRIXINPUT_NROWS) + colInde
    ex_getset_data.matrices.matrixInput[rowIndexSetInput] = value;
}

double get_matrixParam(int colIndex)
{
    int rowIndexGetParam = MATRIXPARAM_NCOLS * (colIndex % MATRIXPARAM_NROWS) + colInde
    return ex_getset_data.matrices.matrixParam[rowIndexGetParam];
}

void set_matrixParam(int colIndex, double value)
{
```

```
    int rowIndexSetParam = MATRIXPARAM_NCOLS * (colIndex % MATRIXPARAM_NROWS) + colInde
    ex_getset_data.matrices.matrixParam[rowIndexSetParam] = value;
}

double get_matrixOutput(int colIndex)
{
    int rowIndexGetOut = MATRIXOUTPUT_NCOLS * (colIndex % MATRIXOUTPUT_NROWS) + colInde
    return ex_getset_data.matrices.matrixOutput[rowIndexGetOut];
}

void set_matrixOutput(int colIndex, double value)
{
    int rowIndexSetOut = MATRIXOUTPUT_NCOLS * (colIndex % MATRIXOUTPUT_NROWS) + colInde
    ex_getset_data.matrices.matrixOutput[rowIndexSetOut] = value;
}
```

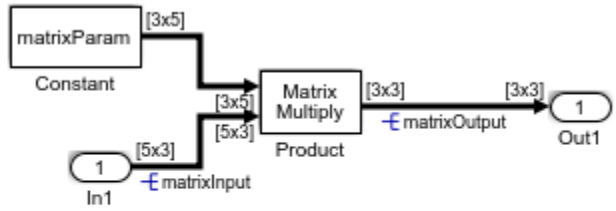
The code that you generate from a model represents matrices as serial arrays. Therefore, each of the `get` and `set` functions accept a single scalar index argument.

The generated code uses column-major format to store and to access matrix data. However, many C applications use row-major indexing. To integrate the generated code with the example legacy code, which stores the matrices `matrixInput` and `matrixParam` using row-major format, the custom `get` functions use the column-major index input to calculate an equivalent row-major index. The generated code algorithm, which interprets matrix data using column-major format by default, performs the correct matrix math because the `get` functions effectively convert the legacy matrices to column-major format. The `set` function for the output, `matrixOutput`, also calculates a row-major index so the code writes the algorithm output to `matrixOutput` using row-major format. Alternatively, to integrate the column-major generated code with your row-major legacy code, you can manually convert the legacy code to column-major format by transposing your matrix data and algorithms.

View the example legacy header file getsetHdrMatrix.h. The file contains the `extern` prototypes for the `get` and `set` functions defined in `getsetSrc.c`.

Open the example model rtwdemo_getset_matrix. The model creates the data objects `matrixInput`, `matrixParam`, and `matrixOutput` in the base workspace. The objects correspond to the signals and parameter in the model.

```
load_system('rtwdemo_getset_matrix')
set_param('rtwdemo_getset_matrix','SimulationCommand','Update')
open_system('rtwdemo_getset_matrix')
```



In the base workspace, double-click the object `matrixInput` to view its properties. The object uses the custom storage class `GetSet`. The property `HeaderFile` is specified as `getsetHdrMatrix.h`. This legacy header file contains the `get` and `set` function prototypes.

In the model Configuration Parameters dialog box, on the **Code Generation > Custom Code** pane, the example legacy source file `getsetSrc.c` is identified for inclusion during the build process. This legacy source file contains the `get` and `set` function definitions and the definition of the global structure variable `ex_getset_data`.

Generate code with the example model.

```
rtwbuild('rtwdemo_getset_matrix');
```

```
### Starting build procedure for model: rtwdemo_getset_matrix
### Successful completion of build procedure for model: rtwdemo_getset_matrix
```

In the code generation report, view the file `rtwdemo_getset_matrix.c`. The model `step` function uses the legacy `get` and `set` functions to execute the algorithm.

```
rtwdemodbtype(fullfile('rtwdemo_getset_matrix_ert_rtw',...
    'rtwdemo_getset_matrix.c'),'/* Model step function */','}',1,1)
```

```
/* Model step function */
void rtwdemo_getset_matrix_step(void)
{
  int32_T i;
  int32_T i_0;
  int32_T i_1;
```

```
/* Product: '<Root>/Product' incorporates:
 *  Constant: '<Root>/Constant'
 *  Inport: '<Root>/In1'
 */
for (i_0 = 0; i_0 < 3; i_0++) {
  for (i = 0; i < 3; i++) {
    set_matrixOutput( i + 3 * i_0 , 0.0);
    for (i_1 = 0; i_1 < 5; i_1++) {
      set_matrixOutput( i + 3 * i_0 , get_matrixParam( 3 * i_1 + i ) *
                        get_matrixInput( 5 * i_0 + i_1 ) + get_matrixOutput( 3 *
        i_0 + i ));
    }
```

Specify Header File or Function Naming Scheme for All Data Items

By default, you specify a header file name, `get` function name, and `set` function name for each data item, such as a signal or parameter, that uses the custom storage class `GetSet`.

To configure a single header file, `get` function naming scheme, or `set` function naming scheme to use for every data item, you can use the Custom Storage Class Designer to create your own copy of `GetSet`. You can specify the header file or function names in a single location.

Follow these steps to create your own custom storage class by creating your own data class package, creating a copy of `GetSet`, and applying the new custom storage class to data items in your model.

1. Create your own data class package, `myPackage`, by copying the example package folder `+SimulinkDemos` and renaming the copied folder as `+myPackage`. Modify the `Parameter` and `Signal` class definitions so that they use the custom storage class definitions from `myPackage`. For an example, see “Create Data Class Package” on page 23-73.
2. Set your current folder to the folder that contains the package folder. Alternatively, add the folder to your MATLAB path.
3. Open the Custom Storage Class Designer.

   ```
   cscdesigner('myPackage')
   ```
4. Select the custom storage class `GetSet`. Click **Copy** to create a copy called `GetSet_1`.

5. Select the new custom storage class `GetSet_1`. In the **General** tab, set **Name** to `myGetSet`.
6. Set the drop-down list **Header file** to `Specify`. In the new text box, set **Header file** to `myFcnHdr.h`. Click **Apply**.
7. On the **Access Function Attributes** tab, set the drop-down lists **Get function** and **Set function** to `Specify`.
8. In the new boxes, set **Get function** to `myGetFcn_$N` and **Set function** to `mySetFcn_$N`. Click **OK**. Click **Yes** in response to the message about saving changes.

   When you generate code, the token `$N` expands into the name of the data item that uses this custom storage class.
9. Apply the custom storage class `myGetSet` from your package to a data item. For example, create a `myPackage.Parameter` object in the base workspace.

   ```
   myParam = myPackage.Parameter(15.23);
   myParam.CoderInfo.StorageClass = 'Custom';
   myParam.CoderInfo.CustomStorageClass = 'myGetSet';
   ```
10. Use the object to set a parameter value in your model. When you generate code, the code algorithm accesses the parameter through the functions that you specified. The code uses a `#include` directive to include the header file that you specified.

`GetSet` Custom Storage Class Restrictions

- `GetSet` does not support complex signals.
- Multiple data in the same model cannot use the same `GetFunction` or `SetFunction`.
- Some blocks do not directly support `GetSet`.
- Custom S-functions do not directly support `GetSet`.

To use `GetSet` with an unsupported block or a custom S-function:

1. Insert a Signal Conversion block at the output of the block or function.
2. In the Signal Conversion block dialog box, select **Exclude this block from 'Block reduction' optimization**.
3. Assign the custom storage class `GetSet` to the output of the Signal Conversion block.

Related Examples

- "Exchange Data Between External C/C++ Code and Simulink Model or Generated Code" on page 39-86
- "Control Data Code by Creating Custom Storage Class" on page 23-73
- "Control Data Representation by Applying Custom Storage Classes" on page 23-58
- "Introduction to Custom Storage Classes" on page 23-2
- "Define Advanced Custom Storage Classes Types" on page 23-78
- "Generate Code That Dereferences Data from a Literal Memory Address" on page 23-83

Configure Generated Code According to Interface Control Document

Import specifications from an interface control document (ICD), configure code generation settings for a model according to the specifications, and store the settings in data dictionaries.

An ICD describes the data interface between two software components. To exchange and share data, the components declare and define global variables that store signal and parameter values. The ICD names the variables and lists characteristics such as data type, physical units, and parameter values. When you create models of the components in Simulink, you can configure the generated code to conform to the interface specification.

In this example, the ICD is a Microsoft® Excel® workbook.

Explore Interface Control Document

Navigate to the folder *`matlabroot`*`/examples/ecoder` (open). Copy these files to a writable, working folder:

- `ICD.xls`
- `importICD.m`

In Microsoft® Excel® or another compatible program, open the `ICD.xls` workbook and view the first worksheet, `Signals`. Each row of the worksheet describes a signal that crosses the interface boundary.

Inspect the cell values in the worksheet. The `Owner` column indicates the name of the component that allocates memory for each signal. The `DataType` column indicates the signal data type in memory. For example, the worksheet uses the expression `Bus: EngSensors` to represent a structure type named `EngSensors`.

In the `Parameters` worksheet, the `Value` column indicates the value of each parameter. If the value of the parameter is nonscalar, the value is stored in its own separate worksheet, which has the same name as the parameter.

In the `Numeric Types` worksheet, each row represents a named numeric data type. In this ICD, the data use fixed-point data types (Fixed-Point Designer). The `IsAlias` column indicates whether the C code uses the name of the data type (for example, `u8En7`)

or uses the name of the primitive integer data type that corresponds to the word length. The `DataScope` column indicates whether the generated code exports or imports the definition of the type.

In the `Structure Types` worksheet, each row represents either a structure type or a field of a structure type. For structure types, the value in the `DataType` column is `struct`. Subsequent rows that do not use `struct` represent fields of the preceding structure type. This ICD defines a structure type, `EngSensors`, with four fields: `throttle`, `speed`, `ego`, and `map`.

In the `Enumerated Types` worksheet, similar to the `Structure Types` worksheet, each row represents either an enumerated type or an enumeration member. This ICD defines an enumerated type `sldemo_FuelModes`.

Write Custom Code

Some data items in the ICD belong to `other component`, which is a component that exists outside of MATLAB. Create the custom code files that define and declare this external data.

Create the custom source file `inter_sigs.c` in your current folder. This file defines the imported signal `sensors`.

```
#include "inter_sigs.h"

EngSensors sensors;                    /* Instrument measurements. */
```

Create the custom header file `inter_sigs.h` in your current folder.

```
#include "inter_types.h"

extern EngSensors sensors;   /* Instrument measurements. */
```

Create the custom header file `inter_types.h` in your current folder. This file defines the structure type `EngSensors` and numeric data types such as `u8En7`.

```
#ifndef INTER_TYPES_H__
#define INTER_TYPES_H__
```

```
typedef short s16En3;

typedef short s16En7;

typedef unsigned char u8En7;

typedef short s16En15;

/* Structure type for instrument measurements. */
typedef struct {
  /* Throttle angle. */
  s16En3 throttle;

  /* Engine speed. */
  s16En3 speed;

  /* EGO sensors. */
  s16En7 ego;

  /* Manifold pressure. */
  u8En7 map;
} EngSensors;

#endif
```

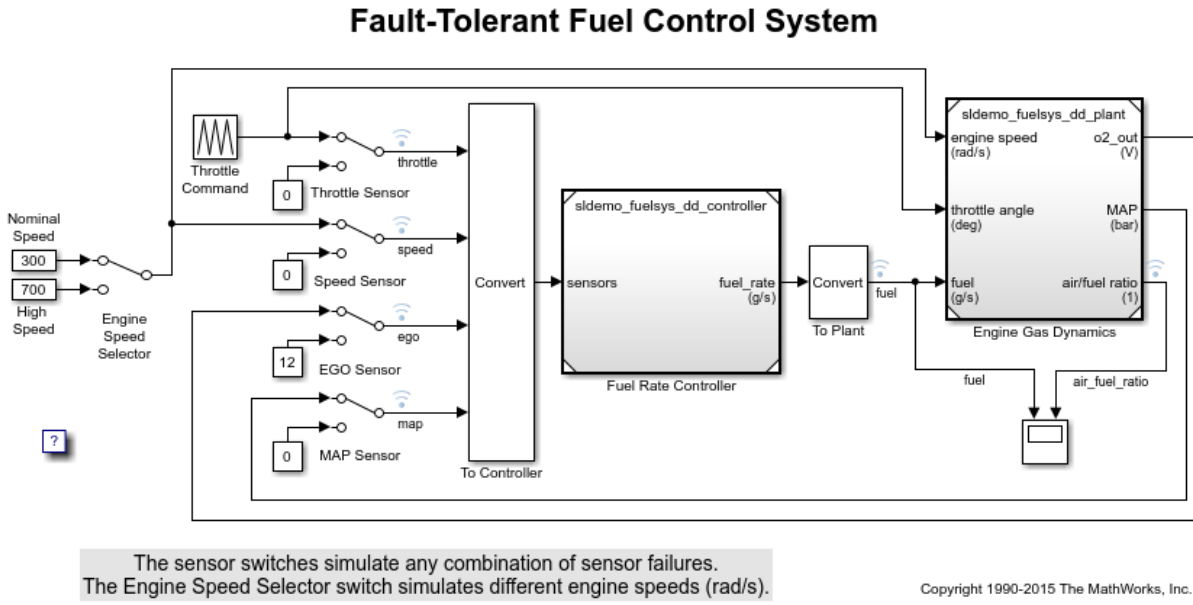
Explore Example Model

Run the script prepare_sldemo_fuelsys_dd. The script prepares a system model, `sldemo_fuelsys_dd`, for this example.

```
run(fullfile(matlabroot,'examples','ecoder','prepare_sldemo_fuelsys_dd'))
```

Open the system model, sldemo_fuelsys_dd.

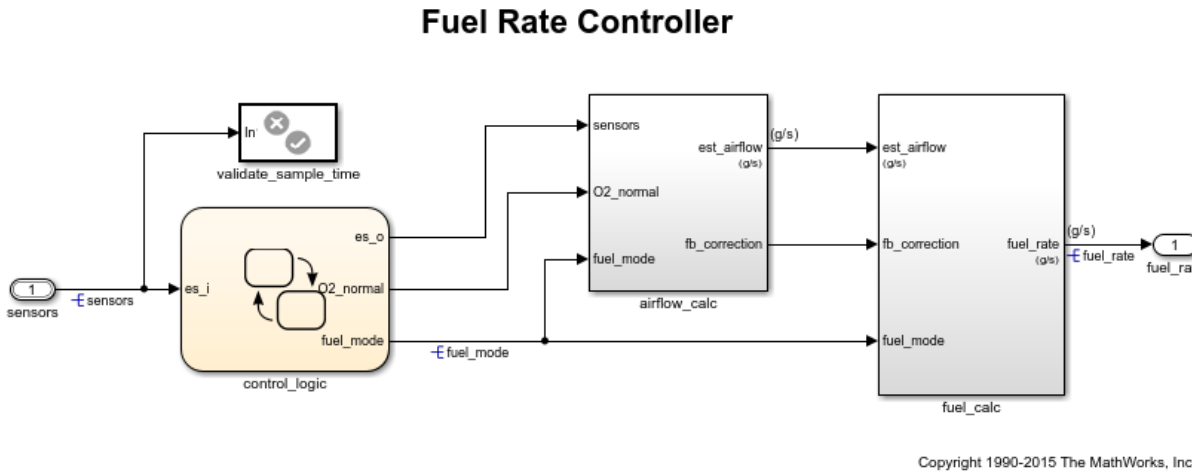
```
sldemo_fuelsys_dd
```



This system model references a controller model. In this example, you generate code from the controller model.

Open the controller model, sldemo_fuelsys_dd_controller.

```
sldemo_fuelsys_dd_controller
```



Data items in the controller model refer to `Simulink.Signal` and `Simulink.Parameter` objects in the base workspace. For example, the input signal `sensors` refers to a `Simulink.Signal` object that has the same name. These objects store settings such as data types, block parameter values, and physical units. The names of these data items and objects match the names of the signals and parameters in the ICD.

Import ICD Specifications into Simulink

To configure code generation settings for the data items, import the settings from the ICD.

Open the example script importICD. The script imports the data from each worksheet of the ICD into variables in the base workspace. It then configures the properties of the `Simulink.Signal` and `Simulink.Parameter` objects in the base workspace by using the imported data.

```
edit('importICD')
```

If the base workspace already contains a data object that corresponds to a target data item in the ICD, the script configures the properties of the existing object. If the object does not exist, the script creates the object.

Run the `importICD` script.

```
run('importICD')
```

The script configures the data objects in the base workspace for code generation according to the specifications in the ICD. The `Simulink.Bus` object `EngSensors` represents the structure type from the ICD. The `Simulink.NumericType` objects, such as `u8En7`, represent the fixed-point data types.

```
ans =

    'Cannot redefine enumerated type sldemo_FuelModes because open models and existing
```

Generate and Inspect Code

Configure the controller model to compile the generated code into an executable by clearing the model configuration parameter **Generate code only**.

Generate code from the controller model.

```
### Starting build procedure for model: sldemo_fuelsys_dd_controller
### Successful completion of build procedure for model: sldemo_fuelsys_dd_controller
```

The generated header file `sldemo_FuelModes.h` defines the enumeration `sldemo_FuelModes`.

```
typedef enum {
  LOW = 1,                              /* Default value */
  RICH,
  DISABLED
} sldemo_FuelModes;
```

The file `sldemo_fuelsys_dd_controller_types.h` includes (`#include`) the custom header file `inter_types.h`, which defines data types such as `u8En7` and the structure type `EngSensors`.

```
#include "inter_types.h"
```

The file `sldemo_fuelsys_dd_controller_private.h` includes the custom header file `inter_sigs.h`. This custom header file contains the `extern` declaration of the signal `sensors`, which a different software component owns.

The data header file `global_data.h` declares the exported parameters and signals that the ICD specifies. To share this data, other components can include this header file.

```
/* Exported data declaration */

/* Declaration for custom storage class: ExportToFile */
extern u8En7 PressEst[855];            /* Lookup table to estimate pressure on sensor f
extern s16En15 PumpCon[551];           /* Lookup table to determine pumping constant ba
extern s16En15 RampRateKiZ[25];        /* Lookup table to determine throttle rate. */
extern s16En3 SpeedEst[1305];          /* Lookup table to estimate engine speed on sens
extern s16En7 ThrotEst[551];           /* Lookup table to estimate throttle angle on se
extern sldemo_FuelModes fuel_mode;     /* Fueling mode of engine. Enrich air/fuel mixtu
extern int16_T fuel_rate;              /* Fuel rate setpoint. */
```

The data definitions (memory allocation) appear in the source files that the ICD specifies, `params.c` and `signals.c`. For example, `params.c` defines and initializes the parameter `RampRateKiZ`.

```
s16En15 RampRateKiZ[25] = { 393, 786, 1180, 1573, 1966, 786, 1573, 2359, 3146,
  3932, 1180, 2359, 3539, 4719, 5898, 1573, 3146, 4719, 6291, 7864, 1966, 3932,
  5898, 7864, 9830 } ;                 /* Lookup table to determine throttle rate. */
```

The algorithm is in the model `step` function in the file `sldemo_fuelsys_dd_controller.c`. The algorithm uses the global data that the ICD identifies. For example, the algorithm uses the value of the signal `fuel_mode` in a `switch` block to control the flow of execution.

```
  /* SwitchCase: '<S10>/Switch Case' incorporates:
   *  Constant: '<S11>/shutoff'
   */
  switch (fuel_mode) {
   case LOW:
    /* Outputs for IfAction SubSystem: '<S10>/low_mode' incorporates:
     *  ActionPort: '<S12>/Action Port'
     */
    /* DiscreteFilter: '<S12>/Discrete Filter' incorporates:
     *  DiscreteIntegrator: '<S1>/Discrete Integrator'
     */
    DiscreteFilter_tmp = (int16_T)(int32_T)((int32_T)((int32_T)((int32_T)
      rtDWork.DiscreteIntegrator_DSTATE << 14) - (int32_T)(-12137 * (int32_T)
      rtDWork.DiscreteFilter_states_g)) >> 14);
```

Change Ownership of Data in ICD

When you make changes to the ICD, you can reuse the `importICD` script to reconfigure the model. Change the ownership of the signal `sensors`, the structure type, and the fixed-point data types from `other_component` to `sldemo_fuelsys_dd_controller`.

In the ICD, on the `signals` worksheet, for the signal `sensors`, set these cell values:

- `Owner` to `sldemo_fuelsys_dd_controller`
- `HeaderFile` to `global_data.h`
- `DefinitionFile` to `signals.c`

On the `Numeric Types` worksheet, for all of the fixed-point data types, set:

- `DataScope` to `Exported`
- `HeaderFile` to `exported_types.h`.

On the `Structure Types` worksheet, for the structure type `EngSensors`, set:

- `DataScope` to `Exported`
- `HeaderFile` to `exported_types.h`.

Rerun the `importICD` script.

```
ans =

    'Cannot redefine enumerated type sldemo_FuelModes because open models and existing
```

Generate code from the model.

```
### Starting build procedure for model: sldemo_fuelsys_dd_controller
### Successful completion of build procedure for model: sldemo_fuelsys_dd_controller
```

The generated file `exported_types.h` defines the structure type `EngSensors` and the fixed-point data types.

```
typedef int16_T s16En3;
typedef int16_T s16En7;
```

```
typedef uint8_T u8En7;

/* Structure type for instrument measurements. */
typedef struct {
  /* Throttle angle. */
  s16En3 throttle;

  /* Engine speed. */
  s16En3 speed;

  /* EGO sensors. */
  s16En7 ego;

  /* Manifold pressure. */
  u8En7 map;
} EngSensors;

typedef int16_T s16En15;
```

The file `signals.c` now includes the definition of the signal `sensors`.

```
/* Exported data definition */

/* Definition for custom storage class: ExportToFile */
sldemo_FuelModes fuel_mode;            /* Fueling mode of engine. Enrich air/fuel mixtu
int16_T fuel_rate;                     /* Fuel rate setpoint. */
EngSensors sensors;                    /* Instrument measurements. */
```

Migrate Base Workspace Data to Data Dictionary

Objects and variables that you create in the base workspace (for example, `Simulink.Parameter` objects) are not saved with the model. When you end your MATLAB session, the objects and variables do not persist. To permanently store the objects and variables, link one or more models to one or more data dictionaries.

Data dictionaries also enable you to track changes made to the objects and variables, which helps you to:

- Reconcile the data stored in MATLAB with the data stored in the ICD.
- Export data from MATLAB to the ICD.

1. In the top model, `sldemo_fuelsys_dd`, select **File > Model Properties > Link to Data Dictionary**.

2. In the Model Properties dialog box, select **Data Dictionary**. Click **New**.
3. In the Create a new Data Dictionary dialog box, set **File name** to `sysDict` and click **Save**.
4. In the Model Properties dialog box, click **OK**.
5. Click **Yes** in response to the message about migrating base workspace data.
6. Click **Yes** in response to the message about removing the imported items from the base workspace.
7. Click **OK** in response to the message about enumerated type migration.

The variables and objects that the models use all exist in the new data dictionary `sysDict.sldd`, which is in your current folder. All three models in the model reference hierarchy are linked to this dictionary.

Create Reference Dictionary

To establish clear ownership of the data that you store in a dictionary, create reference dictionaries.

1. Open the controller model, `sldemo_fuelsys_dd_controller`. Select **File > Model Properties > Link to Data Dictionary**. Click **New**.
2. Set the name of the new dictionary to `ctrlDict.sldd` and click **Save**. In the Model Properties dialog box, click **OK**.
3. In response to the message about changing the dictionary or moving the data, click **Move Data**. Click **Yes** in response to the message about migrating data.

The variables and objects that the controller model uses now exist in the referenced dictionary `ctrlDict.sldd`. Because `sysDict.sldd` references `ctrlDict.sldd`, you can view all of the data by opening `sysDict.sldd` in the Model Explorer.

Now that the model data acquire code generation settings from objects and variables that are stored in data dictionaries, you can modify the `importICD` script so it accesses the dictionaries instead of the base workspace. For more information about the programmatic interface for data dictionaries, see “Store Data in Dictionary Programmatically” (Simulink).

Store Enumerated Type Definition in Data Dictionary

You can import the definition of the enumerated type `sldemo_FuelModes` into the controller dictionary. See “Enumerations in Data Dictionary” (Simulink).

Store Signal and State Design Attributes Inside or Outside of Model File

In this example, you use `Simulink.Signal` objects to specify design attributes such as data types, minimum and maximum values, and physical units. The signal objects store these specifications outside of the model file.

Alternatively, you can store these specifications in the model file by using block and port parameters, which you can access through the Model Data Editor, the Property Inspector, and other dialog boxes.

To decide where to store the specifications, see "Store Design Attributes of Signals and States" (Simulink).

Related Examples

- "Exchange Data Between External C/C++ Code and Simulink Model or Generated Code" on page 39-86
- "Conform to Coding Standards by Replacing and Renaming Data Types" on page 21-22
- "Data Import and Export" (MATLAB)
- "Introduction to Custom Storage Classes" on page 23-2
- "Data Types"
- "What Is a Data Dictionary?" (Simulink)
- "Data Objects" (Simulink)

24

Data Object Wizard in Embedded Coder

Create Data Objects for Code Generation with Data Object Wizard

To specify code generation options for signal lines, block parameters, and states in a model, you can use data objects that you store in a workspace or data dictionary. For basic information about data objects, see “Data Objects” (Simulink).

You can use the Data Object Wizard to create data objects for:

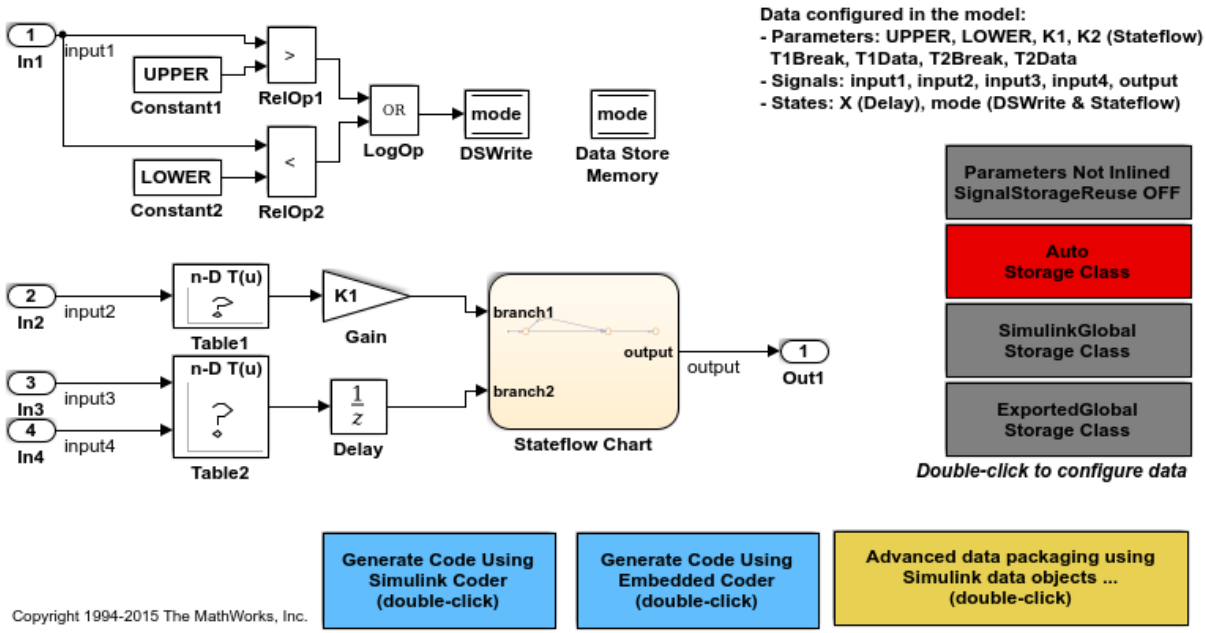
- New or existing models that do not use data objects.
- Existing models to which you have added signal lines or blocks.

This example shows how to use the Data Object Wizard to create and configure data objects for code generation from the built-in package `Simulink`.

Create Data Objects

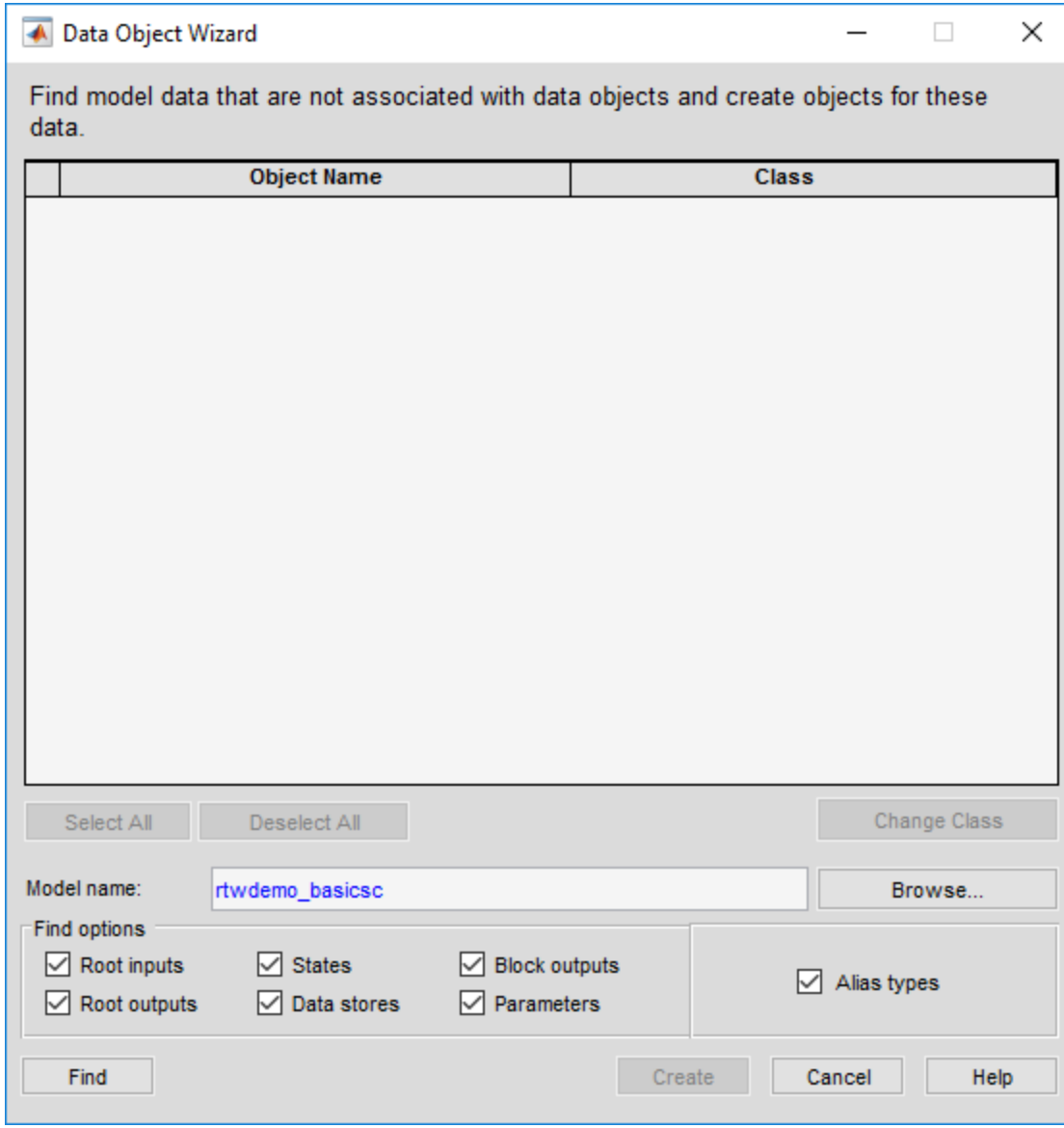
Open the example model rtwdemo_basicsc.

```
open_system('rtwdemo_basicsc')
```

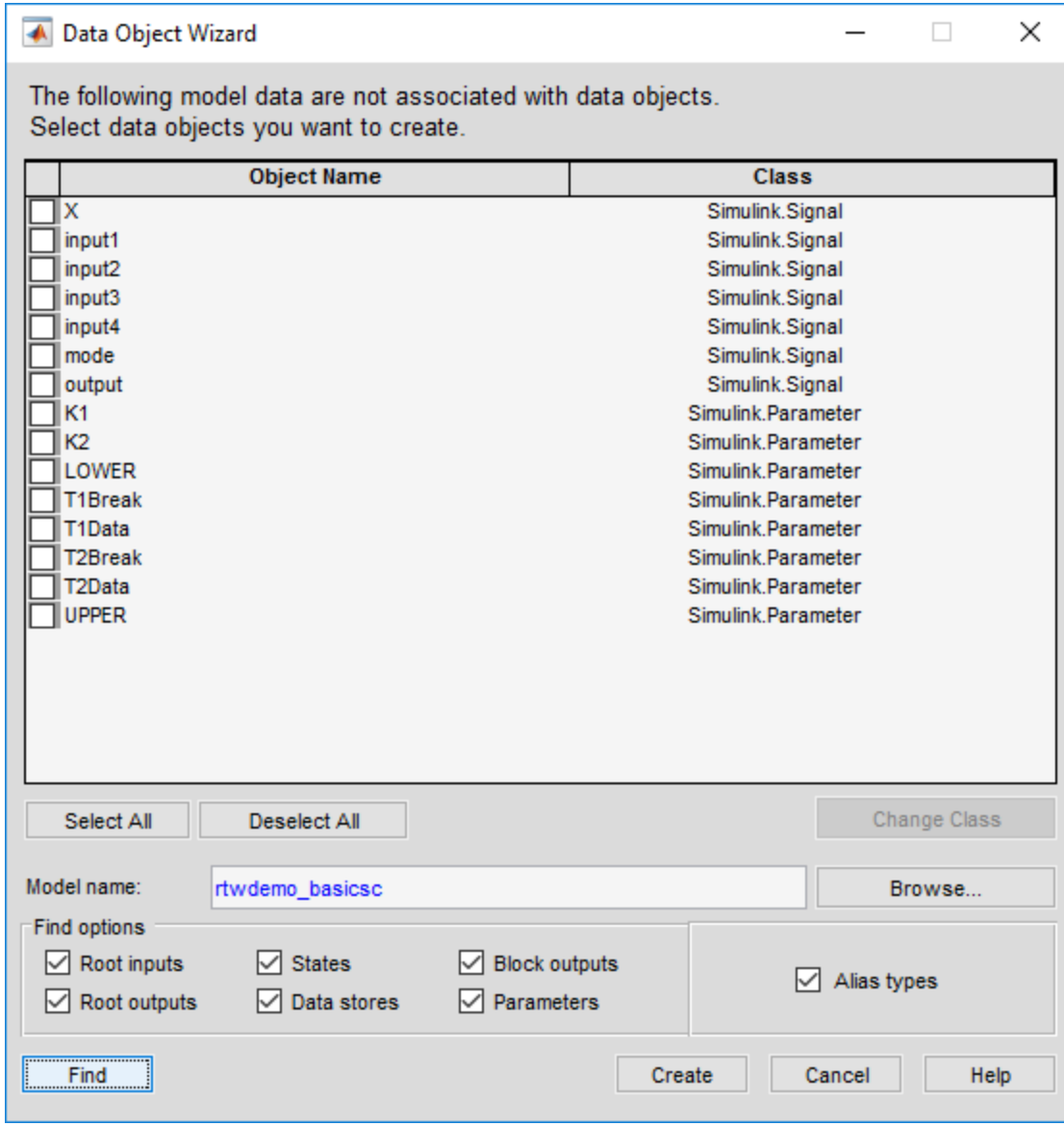


The model creates numeric variables in the base workspace. Blocks in the model use these variables to set parameter values (such as the **Gain** parameter of a Gain block). Some of the signals and block states in the model have explicit names, such as `input1`.

In the model, select **Code > Data Objects > Data Object Wizard**.


Data Object Wizard
Find model data that are not associated with data objects and create objects for these data.
Object Name
Class
Select All
Deselect All
Change Class
Model name:
rtwdemo_basicsc
Browse...
Find options
Root inputs
States
Block outputs
Root outputs
Data stores
Parameters
Alias types
Find
Create
Cancel
Help

In the Data Object Wizard, click **Find**. The wizard proposes the creation of `Simulink.Parameter` objects to replace the variables and the creation of `Simulink.Signal` objects to represent the signals and states.


Data Object Wizard
The following model data are not associated with data objects.
Select data objects you want to create.
Object Name
Class
X
Simulink.Signal
input1
Simulink.Signal
input2
Simulink.Signal
input3
Simulink.Signal
input4
Simulink.Signal
mode
Simulink.Signal
output
Simulink.Signal
K1
Simulink.Parameter
K2
Simulink.Parameter
LOWER
Simulink.Parameter
T1Break
Simulink.Parameter
T1Data
Simulink.Parameter
T2Break
Simulink.Parameter
T2Data
Simulink.Parameter
UPPER
Simulink.Parameter
Select All
Deselect All
Change Class
Model name:
rtwdemo_basicsc
Browse...
Find options
Root inputs
States
Block outputs
Root outputs
Data stores
Parameters
Alias types
Find
Create
Cancel
Help

The wizard finds only signals, parameters, data stores, and states whose storage class is set to `Auto`. For example, if you use the Signal Properties dialog box to specify a storage class other than `Auto` for a signal line, the wizard does not propose a data object.

Click **Select All**.

Click **Create**. The data objects appear in the base workspace.

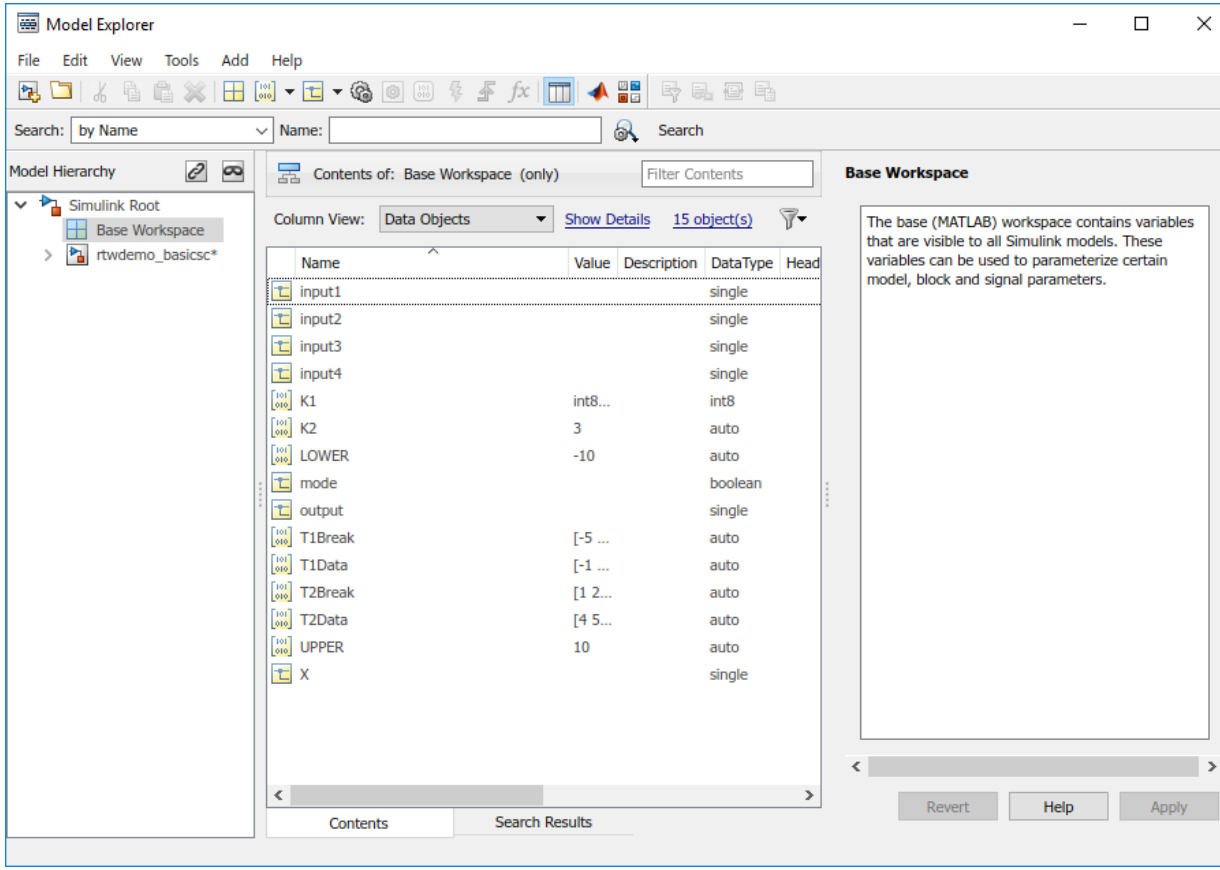
For detailed information about the options that you can choose in the Data Object Wizard, see "Create Data Objects for a Model Using Data Object Wizard" (Simulink).

Set Storage Class for Data Objects

Storage classes determine how the generated code uses variables to represent signals, parameters, and states. For data objects from the built-in package `Simulink`, the default storage class is `Auto`. To specify storage classes for the new data objects, you can use the Model Explorer.

Open the Model Explorer.

In the **Model Hierarchy** pane, select **Base Workspace**.



In the **Contents** pane, from the drop-down list **Column View**, select `Storage Class`.

Select all of the new data objects. For example, select the object `input1`, hold **Shift**, and select the object `X`.

Set the property `StorageClass` for all of the data objects to `ExportToFile`. To change the storage class for all of the selected objects, in the **StorageClass** column, click any of the objects. In the drop-down list, select `ExportToFile`. The change that you make propagates to all of the selected objects.

Specify the `HeaderFile` property for all of the objects as `myExportedHdrFile.h`.

In the model, set **Configuration Parameters > Code Generation > System target file** to `ert.tlc`. With this setting, the code generator honors custom storage classes such as `ExportToFile`.

Generate and Inspect Code

Generate code from the model.

```
### Starting build procedure for model: rtwdemo_basicsc
### Successful completion of build procedure for model: rtwdemo_basicsc
```

In the code generaetion report, view the generated file `myExportedHdrFile.h`. The file contains `extern` declarations for the global variables that correspond to the data objects.

```
/* Exported data declaration */

/* Declaration for custom storage class: ExportToFile */
extern int8_T K1;
extern real_T K2;
extern real32_T LOWER;
extern real32_T T1Break[11];
extern real32_T T1Data[11];
extern real32_T T2Break[3];
extern real32_T T2Data[9];
extern real32_T UPPER;
extern real32_T X;
extern real32_T input1;
extern real32_T input2;
extern real32_T input3;
extern real32_T input4;
extern boolean_T mode;
extern real32_T output;
```

View the file `rtwdemo_basicsc.c`. The file contains the definitions for the global variables. The code assigns numeric values for the variables that correspond to parameter objects.

```
/* Exported data definition */

/* Definition for custom storage class: ExportToFile */
int8_T K1 = 2;
real_T K2 = 3.0;
```

```
real32_T LOWER = -10.0F;
real32_T T1Break[11] = { -5.0F, -4.0F, -3.0F, -2.0F, -1.0F, 0.0F, 1.0F, 2.0F,
  3.0F, 4.0F, 5.0F } ;

real32_T T1Data[11] = { -1.0F, -0.99F, -0.98F, -0.96F, -0.76F, 0.0F, 0.76F,
  0.96F, 0.98F, 0.99F, 1.0F } ;

real32_T T2Break[3] = { 1.0F, 2.0F, 3.0F } ;

real32_T T2Data[9] = { 4.0F, 16.0F, 10.0F, 5.0F, 19.0F, 18.0F, 6.0F, 20.0F,
  23.0F } ;

real32_T UPPER = 10.0F;
real32_T X;
real32_T input1;
real32_T input2;
real32_T input3;
real32_T input4;
boolean_T mode;
real32_T output;
```

See Also

Simulink.Parameter | `Simulink.Signal`

Related Examples

- “Data Objects” (Simulink)
- “Control Signals and States in Code by Applying Storage Classes” (Simulink Coder)
- “Block Parameter Representation in the Generated Code” (Simulink Coder)
- “Control Data Representation by Applying Custom Storage Classes” on page 23-58

25

Entry-Point Functions and Scheduling in Simulink Coder

Entry-Point Functions and Scheduling

The code generator produces the following entry-point functions for a model:

Function	Description
`model_initialize`	Initialization code in the code generated for a model. Call the function *once* at the start of the application code. Do not use this function to reset the real-time model data structure (rtM).
`model_reset`	Code generated if the model includes a Reset Function block. Call the function from the application code to reset conditions or state.
`model_step`	Generated output and update code for blocks in model. If you clear the model configuration parameter "Single output/update function" (Simulink Coder) (selected by default), instead of producing a *`model`*`_step` function, the code generator produces entry-point functions *`model`*`_output` and *`model`*`_update`.
`model_terminate`	Generated code to call for powering off a system. For ERT-based models, suppress generation of this entry-point function by clearing the model configuration parameter "Terminate function required" (Simulink Coder) (set by default).

The calling interface that the code generator produces for each of these entry-point functions differs depending on the value of the model parameter "Code interface packaging" (Simulink Coder):

- `C++ class` (default for C++ language) — Generates a C++ class interface to model code. The generated interface encapsulates required model data into C++ class attributes and model entry-point functions into C++ class methods.
- `Nonreusable function` (default for C language) — Generates nonreusable code. Model entry-point functions pass `(void)`. The code generator statically allocates global model data structures and produces a set of model entry-entry point functions. External code can access the data structures by calling the entry-point functions.
- `Reusable function` — Generates reusable, multi-instance code that is reentrant, as follows:
  - For a GRT-based model, the generated *`model`*`.c` source file contains an allocation function that dynamically allocates model data for each instance of the model. For an ERT-based model, use the "Use dynamic memory allocation for model

initialization" (Simulink Coder) parameter to control whether the code generator produces an allocation function.

- The generated code passes the real-time model data structure in, by reference, as an argument to *`model`*`_step` and the other model entry-point functions.
- The code generator exports the real-time model data structure in the *`model`*`.h` header file.

For an ERT-based model, you can use the "Pass root-level I/O as" (Simulink Coder) parameter to control how the code generator passes root-level input and output arguments to the reusable model entry-point functions. You can include them in the real-time model data structure that the code generator passes to the functions, passes as individual arguments, or passes as references to an input structure and an output structure.

To call the generated entry-point functions from external code, add an `#include` *`model`*`.h` directive to your code.

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

Note: The function reference pages document the C language default (`Nonreusable function`) calling interface generated for these functions.

More About

- "Generate Reentrant Code from Top-Level Models" (Simulink Coder)
- "Generate C++ Class Interface to Model or Subsystem Code" (Simulink Coder)
- "Execution of Code Generated from a Model" (Simulink Coder)
- "Rapid Prototyping Model Functions" (Simulink Coder)
- "Generate Code That Responds to Initialize, Reset, and Terminate Events" (Simulink Coder)
- "Select a System Target File" on page 30-2

Generate Reentrant Code from Top-Level Models

To generate reentrant multi-instance code from a model, select `Reusable function` code interface packaging. When you select the `Reusable function` code interface for a GRT model:

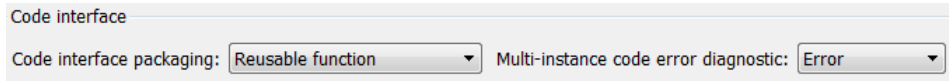
- The generated *`model`*`.c` source file contains an allocation function that dynamically allocates model data for each instance of the model.
- The generated code passes the real-time model data structure in, by reference, as an argument to *`model`*`_step` and the other model entry point functions.
- The real-time model data structure is exported with the *`model`*`.h` header file.

To configure a model to generate reusable, reentrant function code:

1. In the **Code Generation** > **Interface** pane of the Configuration Parameters dialog box, set **Code interface packaging** (Simulink Coder) to the value `Reusable function`. This action enables the parameter **Multi-instance code error diagnostic**.

   Note: If you have an Embedded Coder license and you have selected an ERT target for your model, selecting `Reusable function` enables additional parameters for customizing the generated reusable function interface to model code — **Pass root-level I/O as** (Simulink Coder) and **Use dynamic memory allocation for model initialization** (Simulink Coder).

2. Examine the setting of **Multi-instance code error diagnostic** (Simulink Coder). Leave the parameter at its default value `Error` unless you have a specific need to alter the severity level for diagnostics displayed when a model violates requirements for generating multi-instance code.

3. Generate model code.
4. Examine the model entry-point function interfaces in the generated files and the HTML code generation report. For more information about generating and calling model entry-point functions, see “Entry-Point Functions and Scheduling” (Simulink Coder).

For an example of a model configured to generate reusable, reentrant code, open the example model `rtwdemo_reusable`. To generate GRT code for the example model, double-click the button **Generate Code Using Simulink Coder**.

More About

- "Entry-Point Functions and Scheduling" (Simulink Coder)
- "Generate C++ Class Interface to Model or Subsystem Code" (Simulink Coder)
- "Execution of Code Generated from a Model" (Simulink Coder)
- "Rapid Prototyping Model Functions" (Simulink Coder)

Generate C++ Class Interface to Model or Subsystem Code

On the Configuration Parameters dialog box, set the **Code Generation** > **Interface** "Code interface packaging" (Simulink Coder) parameter to `C++ class` for generating a C++ class interface to model code. The generated interface encapsulates required model data into C++ class attributes and model entry point functions into C++ class methods. The benefits of C++ class encapsulation include:

- Greater control over access to model data
- Ability to multiply instantiate model classes
- Easier integration of model code into C++ programming environments

C++ class encapsulation also works for right-click builds of nonvirtual subsystems. (For information on requirements that apply, see "Generate C++ Class Interface to Nonvirtual Subsystem Code" on page 25-7.)

Generate C++ Class Interface to Model Code

To generate encapsulated C++ class code from a GRT-based model:

1. Set the Configuration Parameters dialog box parameter **Code Generation** > **Language** to `C++`. This selection also enables C++ class code interface packaging for the model.
2. On the **Code Generation** > **Interface** pane, verify that the parameter **Code interface packaging** (Simulink Coder) is set to `C++ class`.
3. Examine the setting of **Multi-instance code error diagnostic** (Simulink Coder). Leave the parameter at its default value `Error` unless you have a specific need to alter the severity level for diagnostics displayed when a model violates requirements for generating multi-instance code.

4. Generate code for the model.
5. Examine the C++ model class code in the generated files *`model`*`.h` and *`model`*`.cpp`. For example, the following code excerpt from the H file generated for the example model `rtwdemo_secondOrderSystem` shows the C++ class declaration for the model.

```
/* Class declaration for model rtwdemo_secondOrderSystem */
```

```
class rtwdemo_secondOrderSystemModelClass {
  /* public data and function members */
 public:
  /* External outputs */
  ExtY_rtwdemo_secondOrderSyste_T rtwdemo_secondOrderSystem_Y;

  /* Model entry point functions */

  /* model initialize function */
  void initialize();

  /* model step function */
  void step();

  /* model terminate function */
  void terminate();

  /* Constructor */
  rtwdemo_secondOrderSystemModelClass();

  /* Destructor */
  ~rtwdemo_secondOrderSystemModelClass();

  /* Real-Time Model get method */
  RT_MODEL_rtwdemo_secondOrderS_T * getRTM();
...
};
```

For more information about generating and calling model entry-point functions, see "Entry-Point Functions and Scheduling" (Simulink Coder).

Note: If you have an Embedded Coder license and you have selected an ERT target for your model, you can use additional **Code Generation** > **Interface** pane parameters to customize the generated C++ class interface to model code.

Generate C++ Class Interface to Nonvirtual Subsystem Code

You can generate C++ class interfaces for right-click builds of nonvirtual subsystems in Simulink models, if the following requirements are met:

- The model is configured for the `C++` language and `C++ class` code interface packaging.
- The subsystem is convertible to a Model block using the function `Simulink.SubSystem.convertToModelReference`. For referenced model conversion requirements, see the Simulink reference page `Simulink.SubSystem.convertToModelReference`.

To configure C++ class interfaces for a subsystem that meets the requirements:

1. Open the containing model and select the subsystem block.
2. Right-click the subsystem and select **C/C++ Code** > **Build This Subsystem**.
3. When the subsystem build completes, examine the C++ class interfaces in the generated files and the HTML code generation report. For more information about generating and calling model entry-point methods, see "Entry-Point Functions and Scheduling" (Simulink Coder).

Note: If you have an Embedded Coder license and you have selected an ERT target for your model, you can use the MATLAB command `RTW.configSubsystemBuild` to customize the generated C++ class interface to subsystem code.

C++ Class Interface Limitations

- Among the data exchange interfaces available on the **Interface** pane of the Configuration Parameters dialog box, only the C API interface is supported for `C++ class` code generation. If you select **External mode** or **ASAP2 interface**, code generation fails with a validation error.
- If a model root inport value connects to a Simscape conversion block, you must insert a Simulink Signal Conversion block between the root inport and the Simscape conversion block. On the Simulink Signal Conversion block parameter dialog box, select **Exclude this block from 'Block reduction' optimization**.
- When building a referenced model that is configured to generate a C++ class interface, you cannot use a C++ class interface in cases when a referenced model cannot have a combined output/update function. Cases include a model that
  - Has a continuous sample time
  - Saves states

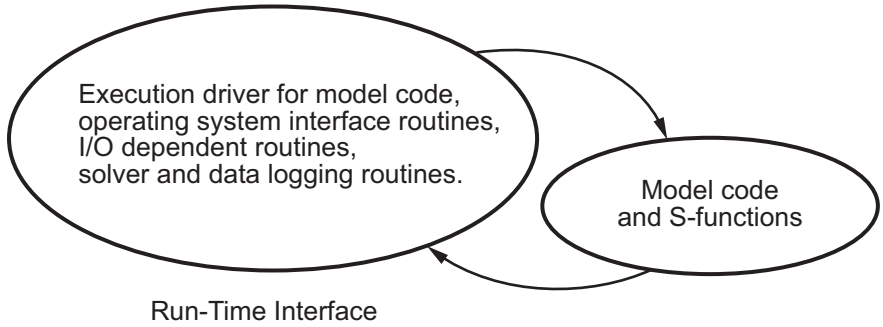
More About

- "Generate Reentrant Code from Top-Level Models" (Simulink Coder)
- "Entry-Point Functions and Scheduling" (Simulink Coder)
- "Execution of Code Generated from a Model" (Simulink Coder)
- "Rapid Prototyping Model Functions" (Simulink Coder)

Execution of Code Generated from a Model

The code generator produces algorithmic code as defined by your model. You can include external (for example, custom or legacy) code in a model by using techniques explained in "Choose an External Code Integration Workflow" (Simulink Coder).

The code generator also provides a interface that executes the generated model code. The interface and model code are compiled together to create an executable program. The next figure shows a high-level object-oriented view of the executable.



The Object-Oriented View of a Real-Time Program

In general, the conceptual design of the model execution driver does not change between the rapid prototyping and embedded style of generated code. The following sections describe model execution for single-tasking and multitasking environments both for simulation (non-real-time) and for real time. For most model code, the multitasking environment will provide the most efficient model execution (that is, fastest sample rate).

The following concepts are useful in describing how model code executes.

- **Initialization**: *model*`_initialize` initializes the interface code and the model code.
- **ModelOutputs**: Calls blocks in your model that have a sample hit at the current time and has them produce their output. *model*`_output` can be done in major or minor time steps. In major time steps, the output is a given simulation time step. In minor time steps, the interface integrates the derivatives to update the continuous states.
- **ModelUpdate**: *model*`_update` calls blocks in your model that have a sample hit at the current point in time and has them update their discrete states or similar type objects.

- **ModelDerivatives**: Calls blocks in your model that have continuous states and has them update their derivatives. *model*_derivatives is only called in minor time steps.
- **ModelTerminate**: *model*_terminate terminates the program if it is designed to run for a finite time. It destroys the real-time model data structure, deallocates memory, and can write data to a file.

Program Execution

A real-time program cannot require 100% of the CPU's time. This provides an opportunity to run background tasks during the free time.

Background tasks include operations such as writing data to a buffer or file, allowing access to program data by third-party data monitoring tools, or using Simulink external mode to update program parameters.

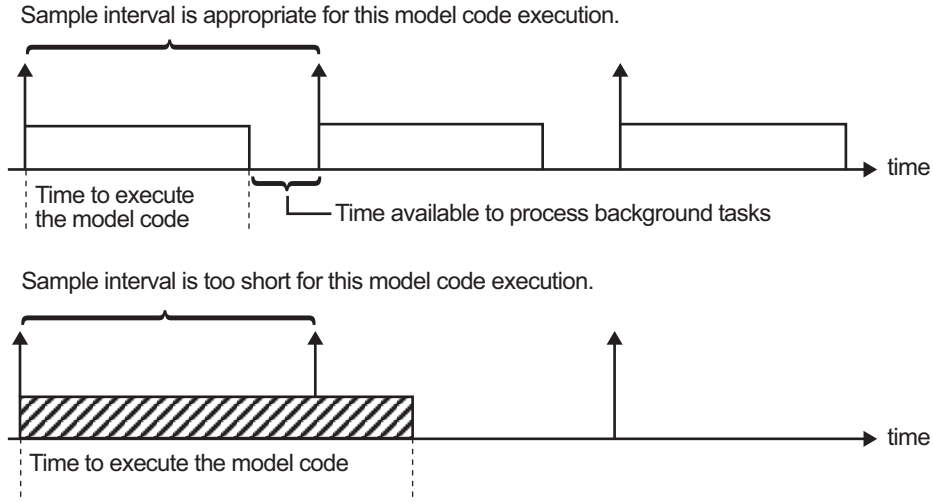
It is important, however, that the program be able to preempt the background task so the model code can execute in real time.

The way the program manages tasks depends on capabilities of the environment in which it operates.

Program Timing

Real-time programs require careful timing of the task invocations (either by using an interrupt or a real-time operating system tasking primitive) so that the model code executes to completion before another task invocation occurs. This includes time to read and write data to and from external hardware.

The next figure illustrates interrupt timing.



Task Timing

The sample interval must be long enough to allow model code execution between task invocations.

In the figure above, the time between two adjacent vertical arrows is the sample interval. The empty boxes in the upper diagram show an example of a program that can complete one step within the interval and still allow time for the background task. The gray box in the lower diagram indicates what happens if the sample interval is too short. Another task invocation occurs before the task is complete. Such timing results in an execution error.

Note also that, if the real-time program is designed to run forever (that is, the final time is 0 or infinite so that the `while` loop never exits), then the shutdown code does not execute.

For more information on how the timing engine works, see "Absolute and Elapsed Time Computation" (Simulink Coder).

External Mode Communication

External mode allows communication between the Simulink block diagram and the standalone program that is built from the generated code. In this mode, the real-time

program functions as an interprocess communication server, responding to requests from the Simulink engine.

Data Logging in Single-Tasking and Multitasking Model Execution

"Debug" on page 28-23 explains how you can save system states, outputs, and time to a MAT-file at the completion of the model execution. The `LogTXY` function, which performs data logging, operates differently in single-tasking and multitasking environments.

If you examine how `LogTXY` is called in the single-tasking and multitasking environments, you will notice that for single-tasking `LogTXY` is called after `ModelOutputs`. During this `ModelOutputs` call, blocks that have a hit at time *t* execute, whereas in multitasking, `LogTXY` is called after `ModelOutputs(tid=0)`, which executes only the blocks that have a hit at time *t* and that have a task identifier of 0. This results in differences in the logged values between single-tasking and multitasking logging. Specifically, consider a model with two sample times, the faster sample time having a period of 1.0 second and the slower sample time having a period of 10.0 seconds. At time t = k*10, k=0,1,2... both the fast (`tid=0`) and slow (`tid=1`) blocks execute. When executing in multitasking mode, when `LogTXY` is called, the slow blocks execute, but the previous value is logged, whereas in single-tasking the current value is logged.

Another difference occurs when logging data in an enabled subsystem. Consider an enabled subsystem that has a slow signal driving the enable port and fast blocks within the enabled subsystem. In this case, the evaluation of the enable signal occurs in a slow task, and the fast blocks see a delay of one sample period; thus the logged values will show these differences.

To summarize differences in logged data between single-tasking and multitasking, differences will be seen when

- Any root outport block has a sample time that is slower than the fastest sample time
- Any block with states has a sample time that is slower than the fastest sample time
- Any block in an enabled subsystem where the signal driving the enable port is slower than the rate of the blocks in the enabled subsystem

For the first two cases, even though the logged values are different between single-tasking and multitasking, the model results are not different. The only real difference is where (at what point in time) the logging is done. The third (enabled subsystem) case results in a delay that can be seen in a real-time environment.

Non-Real-Time Single-Tasking Systems

The pseudocode below shows the execution of a model for a non-real-time single-tasking system.

```
main()
{
  Initialization
  While (time < final time)
    ModelOutputs     -- Major time step.
    LogTXY           -- Log time, states and root outports.
    ModelUpdate      -- Major time step.
    Integrate        -- Integration in minor time step for
                     -- models with continuous states.
      ModelDerivatives
      Do 0 or more
        ModelOutputs
        ModelDerivatives
      EndDo -- Number of iterations depends upon the solver
      Integrate derivatives to update continuous states.
    EndIntegrate
  EndWhile
  Termination
}
```

The initialization phase begins first. This consists of initializing model states and setting up the execution engine. The model then executes, one step at a time. First `ModelOutputs` executes at time *t*, then the workspace I/O data is logged, and then `ModelUpdate` updates the discrete states. Next, if your model has continuous states, `ModelDerivatives` integrates the continuous states' derivatives to generate the states for time $t_{new} = t + h$, where h is the step size. Time then moves forward to t_{new} and the process repeats.

During the `ModelOutputs` and `ModelUpdate` phases of model execution, only blocks that reach the current point in time execute.

Non-Real-Time Multitasking Systems

The pseudocode below shows the execution of a model for a non-real-time multitasking system.

```
main()
```

```
{
  Initialization
  While (time < final time)
    ModelOutputs(tid=0)   -- Major time step.
    LogTXY                -- Log time, states, and root
                          -- outports.
    ModelUpdate(tid=0)    -- Major time step.
    Integrate       -- Integration in minor time step for
                    -- models with continuous states.
      ModelDerivatives
      Do 0 or more
        ModelOutputs(tid=0)
        ModelDerivatives
      EndDo (Number of iterations depends upon the solver.)
      Integrate derivatives to update continuous states.
    EndIntegrate
    For i=1:NumTids
      ModelOutputs(tid=i) -- Major time step.
      ModelUpdate(tid=i)  -- Major time step.
    EndFor
  EndWhile
  Termination
  }
```

Multitasking operation is more complex than single-tasking execution because the output and update functions are subdivided by the *task identifier* (`tid`) that is passed into these functions. This allows for multiple invocations of these functions with different task identifiers using overlapped interrupts, or for multiple tasks when using a real-time operating system. In simulation, multiple tasks are emulated by executing the code in the order that would occur if no preemption existed in a real-time system.

Multitasking execution assumes that all task rates are multiples of the base rate. The Simulink product enforces this when you create a fixed-step multitasking model. The multitasking execution loop is very similar to that of single-tasking, except for the use of the task identifier (`tid`) argument to `ModelOutputs` and `ModelUpdate`.

Note: You cannot use `tid` values from code generated by a target file and not by Simulink Coder. Simulink Coder tracks the use of `tid` when generating code for a specific subsystem or function type. When you generate code in a target file, this argument cannot be tracked because the scope does not have subsystem or function type. Therefore, `tid` becomes an undefined variable and your target file fails to compile.

Real-Time Single-Tasking Systems

The pseudocode below shows the execution of a model in a real-time single-tasking system where the model is run at interrupt level.

```
rtOneStep()
{
  Check for interrupt overflow
  Enable "rtOneStep" interrupt
  ModelOutputs    -- Major time step.
  LogTXY          -- Log time, states and root outports.
  ModelUpdate     -- Major time step.
  Integrate       -- Integration in minor time step for models
                  -- with continuous states.
     ModelDerivatives
     Do 0 or more
       ModelOutputs
       ModelDerivatives
     EndDo (Number of iterations depends upon the solver.)
     Integrate derivatives to update continuous states.
  EndIntegrate
}

main()
{
  Initialization (including installation of rtOneStep as an
  interrupt service routine, ISR, for a real-time clock).
  While(time < final time)
    Background task.
  EndWhile
  Mask interrupts (Disable rtOneStep from executing.)
  Complete any background tasks.
  Shutdown
}
```

Real-time single-tasking execution is very similar to non-real-time single-tasking execution, except that instead of free-running the code, the `rt_OneStep` function is driven by a periodic timer interrupt.

At the interval specified by the program's base sample rate, the interrupt service routine (ISR) preempts the background task to execute the model code. The base sample rate is the fastest in the model. If the model has continuous blocks, then the integration step size determines the base sample rate.

For example, if the model code is a controller operating at 100 Hz, then every 0.01 seconds the background task is interrupted. During this interrupt, the controller reads its inputs from the analog-to-digital converter (ADC), calculates its outputs, writes these outputs to the digital-to-analog converter (DAC), and updates its states. Program control then returns to the background task. All of these steps must occur before the next interrupt.

Real-Time Multitasking Systems

The following pseudocode shows how a model executes in a real-time multitasking system where the model is run at interrupt level.

```
rtOneStep()
{
  Check for interrupt overflow
  Enable "rtOneStep" interrupt
  ModelOutputs(tid=0)     -- Major time step.
  LogTXY                  -- Log time, states and root outports.
  ModelUpdate(tid=0)      -- Major time step.
  Integrate               -- Integration in minor time step for
                          -- models with continuous states.
     ModelDerivatives
     Do 0 or more
       ModelOutputs(tid=0)
       ModelDerivatives
     EndDo (Number of iterations depends upon the solver.)
     Integrate derivatives and update continuous states.
  EndIntegrate
  For i=1:NumTasks
    If (hit in task i)
      ModelOutputs(tid=i)
      ModelUpdate(tid=i)
    EndIf
  EndFor
}

main()
{
  Initialization (including installation of rtOneStep as an
    interrupt service routine, ISR, for a real-time clock).
  While(time < final time)
    Background task.
  EndWhile
```

```
  Mask interrupts (Disable rtOneStep from executing.)
  Complete any background tasks.
  Shutdown
}
```

Running models at interrupt level in a real-time multitasking environment is very similar to the previous single-tasking environment, except that overlapped interrupts are employed for concurrent execution of the tasks.

The execution of a model in a single-tasking or multitasking environment when using real-time operating system tasking primitives is very similar to the interrupt-level examples discussed above. The pseudocode below is for a single-tasking model using real-time tasking primitives.

```
tSingleRate()
{
  MainLoop:
    If clockSem already "given", then error out due to overflow.
    Wait on clockSem
    ModelOutputs            -- Major time step.
    LogTXY                  -- Log time, states and root
                            -- outports
    ModelUpdate             -- Major time step
    Integrate               -- Integration in minor time step
                            -- for models with continuous
                            -- states.
      ModelDeriviatives
      Do 0 or more
        ModelOutputs
        ModelDerivatives
      EndDo (Number of iterations depends upon the solver.)
      Integrate derivatives to update continuous states.
    EndIntegrate
  EndMainLoop
}

main()
{
  Initialization
  Start/spawn task "tSingleRate".
  Start clock that does a "semGive" on a clockSem semaphore.
  Wait on "model-running" semaphore.
  Shutdown
}
```

In this single-tasking environment, the model executes as real-time operating system tasking primitives. In this environment, create a single task (`tSingleRate`) to run the model code. This task is invoked when a clock tick occurs. The clock tick gives a `clockSem` (clock semaphore) to the model task (`tSingleRate`). The model task waits for the semaphore before executing. The clock ticks occur at the fundamental step size (base rate) for your model.

Multitasking Systems Using Real-Time Tasking Primitives

The pseudocode below is for a multitasking model using real-time tasking primitives.

```
tSubRate(subTaskSem,i)
{
  Loop:
    Wait on semaphore subTaskSem.
    ModelOutputs(tid=i)
    ModelUpdate(tid=i)
  EndLoop
}
tBaseRate()
{
  MainLoop:
    If clockSem already "given", then error out due to overflow.
    Wait on clockSem
    For i=1:NumTasks
      If (hit in task i)
        If task i is currently executing, then error out due to
          overflow.
        Do a "semGive" on subTaskSem for task i.
      EndIf
    EndFor
    ModelOutputs(tid=0)    -- major time step.
    LogTXY                 -- Log time, states and root outports.
    ModelUpdate(tid=0)     -- major time step.
    Loop:                  -- Integration in minor time step for
                           -- models with continuous states.
      ModelDeriviatives
      Do 0 or more
        ModelOutputs(tid=0)
        ModelDerivatives
      EndDo (number of iterations depends upon the solver).
      Integrate derivatives to update continuous states.
    EndLoop
```

```
  EndMainLoop
}
main()
{
  Initialization
  Start/spawn task "tSubRate".
  Start/spawn task "tBaseRate".

  Start clock that does a "semGive" on a clockSem semaphore.
  Wait on "model-running" semaphore.
  Shutdown
}
```

In this multitasking environment, the model is executed using real-time operating system tasking primitives. Such environments require several model tasks (`tBaseRate` and several `tSubRate` tasks) to run the model code. The base rate task (`tBaseRate`) has a higher priority than the subrate tasks. The subrate task for `tid=1` has a higher priority than the subrate task for `tid=2`, and so on. The base rate task is invoked when a clock tick occurs. The clock tick gives a `clockSem` to `tBaseRate`. The first thing `tBaseRate` does is give semaphores to the subtasks that have a hit at the current point in time. Because the base rate task has a higher priority, it continues to execute. Next it executes the fastest task (`tid=0`), consisting of blocks in your model that have the fastest sample time. After this execution, it resumes waiting for the clock semaphore. The clock ticks are configured to occur at the fundamental step size for your model.

Rapid Prototyping and Embedded Model Execution Differences

The rapid prototyping program framework provides a common application programming interface (API) that does not change between model definitions.

The Embedded Coder product provides a different framework called the embedded program framework. The embedded program framework provides an optimized API that is tailored to your model. When you use the embedded style of generated code, you are modeling how you would like your code to execute in your embedded system. Therefore, the definitions defined in your model should be specific to your embedded targets. Items such as the model name, parameter, and signal storage class are included as part of the API for the embedded style of code.

One major difference between the rapid prototyping and embedded style of generated code is that the latter contains fewer entry-point functions. The embedded style of code can be configured to have only one function, *`model`*`_step`.

Thus, model execution code eliminates `Loop...EndLoop` statements and groups `ModelOutputs`, `LogTXY`, and `ModelUpdate` into a single statement, *`model`*`_step`.

For more information about how generated embedded code executes, see "Entry-Point Functions and Scheduling" on page 25-2.

More About

- "Time-Based Scheduling and Code Generation" on page 16-2
- "Sample Times in Subsystems" (Simulink)
- "Sample Times in Systems" (Simulink)
- "Time-Based Scheduling Example Models" on page 16-36

Rapid Prototyping Model Functions

Rapid prototyping code defines the following functions that interface with the main program (`main.c` or `main.cpp`):

- `Model()`: The model registration function. This function initializes the work areas (for example, allocating and setting pointers to various data structures) used by the model. The model registration function calls the `MdlInitializeSizes` and `MdlInitializeSampleTimes` functions. These two functions are very similar to the S-function `mdlInitializeSizes` and `mdlInitializeSampleTimes` methods.
- `MdlStart(void)`: After the model registration functions `MdlInitializeSizes` and `MdlInitializeSampleTimes` execute, the main program starts execution by calling `MdlStart`. This routine is called once at startup.

  The function `MdlStart` has four basic sections:

  - Code to initialize the states for each block in the root model that has states. A subroutine call is made to the "initialize states" routines of conditionally executed subsystems.
  - Code generated by the one-time initialization (start) function for each block in the model.
  - Code to enable the blocks in the root model that have enable methods, and the blocks inside triggered or function-call subsystems residing in the root model. Simulink blocks can have enable and disable methods. An enable method is called just before a block starts executing, and the disable method is called just after the block stops executing.
  - Code for each block in the model whose output value is constant. The block code appears in the `MdlStart` function only if the block parameters are not tunable in the generated code and if the code generator cannot eliminate the block code through constant folding.
- `MdlOutputs(int_T tid)`: `MdlOutputs` updates the output of blocks. The `tid` (task identifier) parameter identifies the task that in turn maps when to execute blocks based upon their sample time. This routine is invoked by the main program during major and minor time steps. The major time steps are when the main program is taking an actual time step (that is, it is time to execute a specific task). If your model contains continuous states, the minor time steps will be taken. The minor time steps are when the solver is generating integration stages, which are points between major outputs. These integration stages are used to compute the derivatives used in advancing the continuous states.

- `MdlUpdate(int_T tid)`: `MdlUpdate` updates the states and work vector state information (that is, states that are neither continuous nor discrete) saved in work vectors. The `tid` (task identifier) parameter identifies the task that in turn indicates which sample times are active, allowing you to conditionally update only states of active blocks. This routine is invoked by the interface after the major `MdlOutputs` has been executed. The solver is also called, and *`model`*`_Derivatives` is called in minor steps by the solver during its integration stages. All blocks that have continuous states have an identical number of derivatives. These blocks are required to compute the derivatives so that the solvers can integrate the states.
- `MdlTerminate(void)`: `MdlTerminate` contains any block shutdown code. `MdlTerminate` is called by the interface, as part of the termination of the real-time program.

The contents of the above functions are directly related to the blocks in your model. A Simulink block can be generalized to the following set of equations.

$$y = f_0(t, x_c, x_d, u)$$

Output y is a function of continuous state x_c, discrete state x_d, and input u. Each block writes its specific equation in a section of `MdlOutputs`.

$$x_{d+1} = f_u(t, x_d, u)$$

The discrete states x_d are a function of the current state and input. Each block that has a discrete state updates its state in `MdlUpdate`.

$$\dot{x} = f_d(t, x_c, u)$$

The derivatives x are a function of the current input. Each block that has continuous states provides its derivatives to the solver (for example, `ode5`) in *`model`*`_Derivatives`. The derivatives are used by the solver to integrate the continuous state to produce the next value.

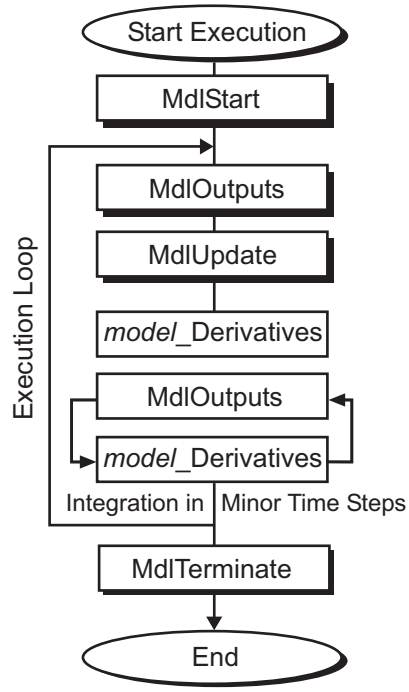
The output, y, is generally written to the block I/O structure. Root-level Outport blocks write to the external outputs structure. The continuous and discrete states are stored in the states structure. The input, u, can originate from another block's output, which is located in the block I/O structure, an external input (located in the external inputs

structure), or a state. These structures are defined in the *model*.h file that the Simulink Coder software generates.

The next example shows the general contents of the rapid prototyping style of C code written to the *model*.c file.

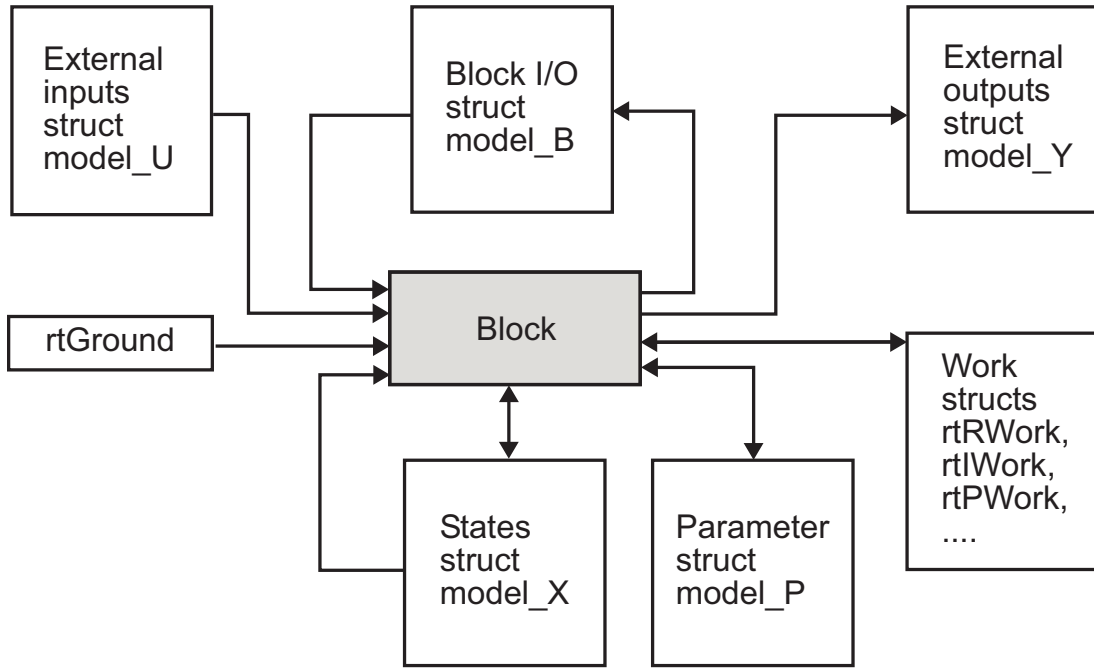
```
/*
 * Version, Model options, TLC options,
 * and code generation information are placed here.
*/
<includes>
void MdlStart(void)
{
  /*
   * State initialization code.
   * Model start-up code - one time initialization code.
   * Execute any block enable methods.
   * Initialize output of any blocks with constant sample times.
   */
}

void MdlOutputs(int_T tid)
{
  /* Compute: y = f0(t,xc,xd,u) for each block as needed. */
}

void MdlUpdate(int_T tid)
{
  /* Compute: xd+1 = fu(t,xd,u) for each block as needed. */

  /* Compute: dxc = fd(t,xc,u) for each block in model_derivatives
                               as needed. */
}
void MdlTerminate(void)
{
  /* Perform shutdown code for any blocks that
     have a termination action */
}
```

The next figure shows a flow chart describing the execution of the rapid prototyping generated code.



Rapid Prototyping Execution Flow Chart

Each block places code in specific `Mdl` routines according to the algorithm that it is implementing. Blocks have input, output, parameters, and states, as well as other general items. For example, in general, block inputs and outputs are written to a block I/O structure (*`model`*`_B`). Block inputs can also come from the external input structure (*`model`*`_U`) or the state structure when connected to a state port of an integrator (*`model`*`_X`), or ground (`rtGround`) if unconnected or grounded. Block outputs can also go to the external output structure (*`model`*`_Y`). The next figure shows the general mapping between these items.



Data View of the Generated Code

The following list defines the structures shown in the preceding figure:

- Block I/O structure (*`model`*`_B`): This structure consists of persistent block output signals. The number of block output signals is the sum of the widths of the data output ports of all nonvirtual blocks in your model. If you activate block I/O optimizations, the Simulink and Simulink Coder products reduce the size of the *`model`*`_B` structure by

  - Reusing the entries in the *`model`*`_B` structure
  - Making other entries local variables

  See “Signal Representation in Generated Code” on page 19-112 for more information on these optimizations.

  Structure field names are determined either by the block's output signal name (when present) or by the block name and port number when the output signal is left unlabeled.

- Block states structures: The continuous states structure (*model*_X) contains the continuous state information for blocks in your model that have continuous states. Discrete states are stored in a data structure called the *DWork vector* (*model*_DWork).
- Block parameters structure (*model*_P): The parameters structure contains block parameters that can be changed during execution (for example, the parameter of a Gain block).
- External inputs structure (*model*_U): The external inputs structure consists of all root-level Inport block signals. Field names are determined by either the block's output signal name, when present, or by the Inport block's name when the output signal is left unlabeled.
- External outputs structure (*model*_Y): The external outputs structure consists of all root-level Outport blocks. Field names are determined by the root-level Outport block names in your model.
- Real work, integer work, and pointer work structures (*model*_RWork, *model*_IWork, *model*_PWork): Blocks might have a need for real, integer, or pointer work areas. For example, the Memory block uses a real work element for each signal. These areas are used to save internal states or similar information.

More About

- "Time-Based Scheduling and Code Generation" on page 16-2

Function and Class Interfaces in Embedded Coder

Control Generation of Function Prototypes

About Function Prototype Control

For fixed-step, rate-based models that you configure with an ERT-based system target file, you can control the prototypes of functions that the code generator produces. Control the function prototype generation by using the **Configure Model Functions** button, located on the **Code Generation** > **Interface** pane of the Configuration Parameters dialog box.

By default, the function prototype of the generated *`model`*`_step` function resembles the following:

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

The function prototype of the generated *`model`*`_initialize` function resembles the following:

```
void model_initialize(void);
```

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

The **Configure Model Functions** button on the **Interface** pane provides you flexible control over the model function prototypes that are generated for your model. Clicking **Configure Model Functions** launches a Model Interface dialog box (see "Configure Function Prototypes Using Graphical Interfaces" on page 26-3). Based on the **Function specification** value you specify for your model function (supported values include `Default model initialize and step functions` and `Model specific C prototypes`), you can preview and modify the function prototypes. Once you validate and apply your changes, you can generate code based on your function prototype modifications.

For more information about using the **Configure Model Functions** button and the Model Interface dialog box, see "Sample Procedure for Configuring Function Prototypes" on page 26-11 and the model `rtwdemo_fcnprotoctrl`, which is preconfigured to demonstrate function prototype control.

Alternatively, you can use function prototype control functions to programmatically control model function prototypes. For more information, see "Configure Function Prototypes Programmatically" on page 26-16.

You can also control model function prototypes for nonvirtual subsystems, if you generate subsystem code using right-click build. To launch the **Model Interface for subsystem** dialog box, use the `RTW.configSubsystemBuild` function.

Right-click building the subsystem generates the step and initialization functions according to the customizations you make. For more information, see "Configure Function Prototypes for Nonvirtual Subsystems" on page 26-9.

For limitations that apply, see "Function Prototype Control Limitations" on page 26-21.

Configure Function Prototypes Using Graphical Interfaces

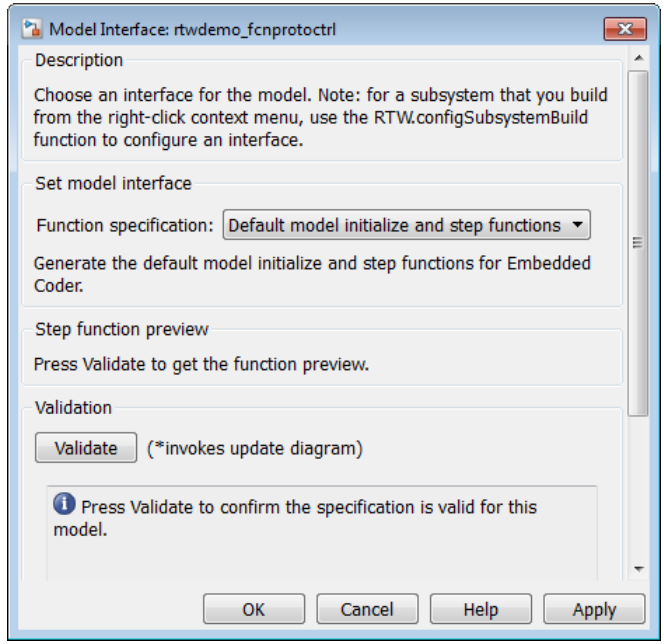
Launch the Model Interface Dialog Boxes

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

To configure function prototypes for a right-click build of a nonvirtual subsystem, invoke the `RTW.configSubsystemBuild` function, which launches the Model Interface for subsystem dialog box. For more information, see "Configure Function Prototypes for Nonvirtual Subsystems" on page 26-9

Default Model Initialize and Step Functions View

The figure below shows the Model Interface dialog box in the `Default model initialize and step functions` view.



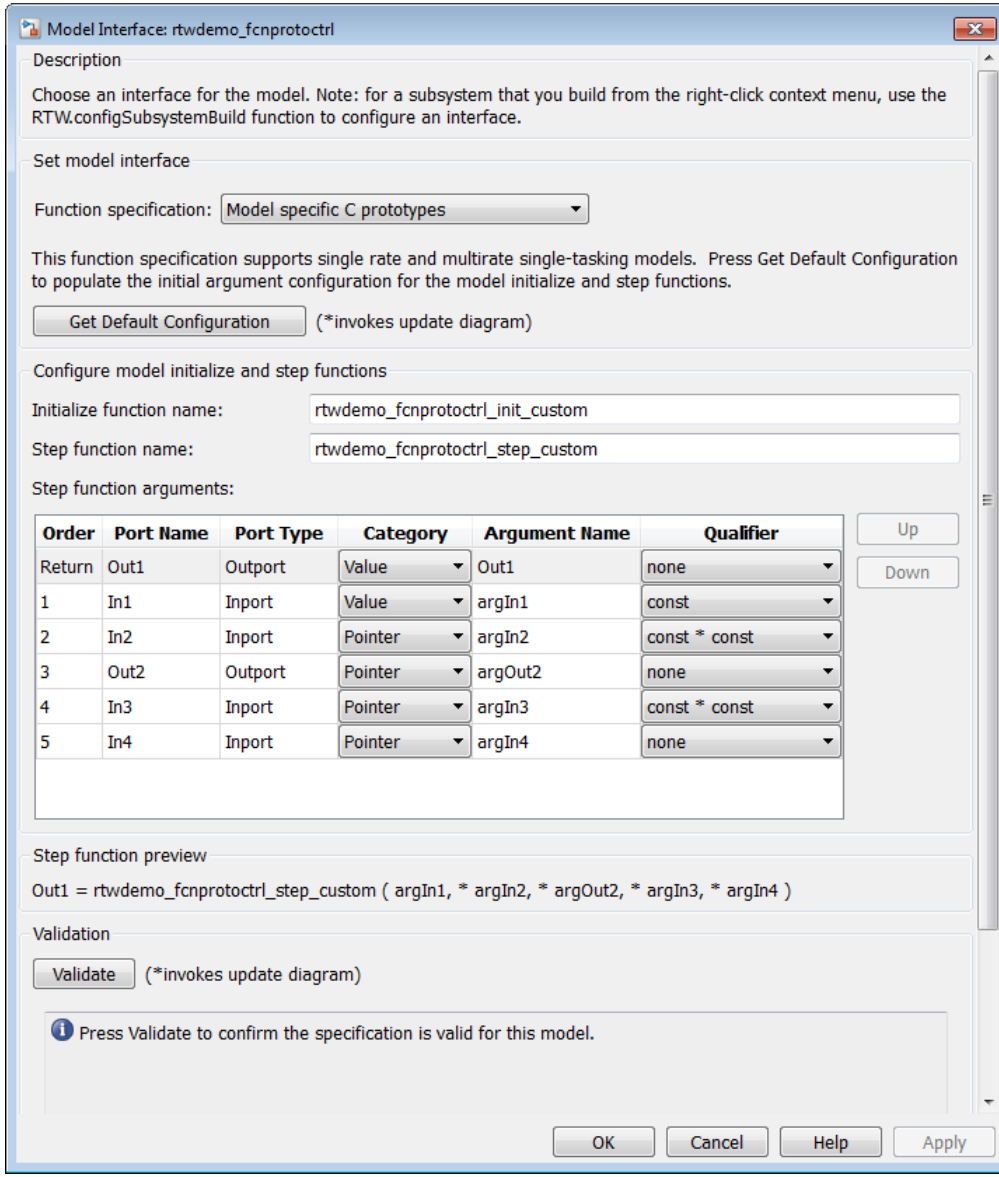
The `Default model initialize and step functions` view allows you to validate and preview the predicted default model step and initialization function prototypes. To validate the default function prototype configuration against your model, click the **Validate** button. If the validation succeeds, the predicted step function prototype appears in the **Step function preview** subpane.

Note: You cannot use the `Default model initialize and step functions` view to modify the function prototype configuration.

Model Specific C Prototypes View

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

To begin configuring your function control prototype configuration, click the **Get Default Configuration** button. This activates and initializes the function names and properties in the **Configure model initialize and step functions** subpane, as shown below. If you click **Get Default Configuration** again later, only the properties of the step function arguments are reset to default values.



In the **Configure model initialize and step functions** subpane:

Parameter	Description
Step function name	Name of the *`model`*`_step` function.
Initialize function name	Name of the *`model`*`_initialize` function. **Note:** A referenced model contains at least one initialization function. When the model is not built as a referenced model, this parameter controls the name of the function that initializes states to nonzero values. A model generates this function only if it contains such states or requires the function for some other less common reason. The code generator determines the names of the other initialization functions. When built as a referenced model, this parameter does not control the name of the `Model Initialize fcn for ModelReference Block`. The code generator determines the name of this function for referenced model builds.
Order	Order of the argument. A return argument is listed as `Return`.
Port Name	Name of the port.
Port Type	Type of the port.
Category	Specifies how an argument is passed in or out from the customized step function, either by copying a value (`Value`) or by a pointer to a memory space (`Pointer`).
Argument Name	Name of the argument.

Parameter	Description
Qualifier (optional)	Specifies a `const` type qualifier for a function argument. The available values are dependent on the **Category** specified. When you change the **Category**, if the specified type is not available, the **Qualifier** changes to `none`. The possible values are: • `none` • `const` (value) • `const*` (value referenced by the pointer) • `const*const` (value referenced by the pointer and the pointer itself) **Note:** When a model includes a referenced model, the `const` type qualifier for the root input argument of the referenced model's specified step function interface is set to `none`, and the qualifier for the source signal in the referenced model's parent is set to a value other than `none`, code generation honors the referenced model's interface specification by generating a type cast that discards the `const` type qualifier from the source signal. To override this behavior, add a `const` type qualifier to the referenced model.

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

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

Note: The list of step function arguments has an entry for each of the model's root-level I/O ports. This list does not include model parameter arguments that can appear in the generated code when the model is used as a referenced model. For example, a model `sldemo_mdlref_counter_paramargs` has an inport with argument name `arg_input`, an outport with argument name `arg_output`, and a saturation block whose

limits have workspace parameter argument names `lower_saturation_limit` and `upper_saturation_limit`.

The step function preview for this model is:

```
sldemo_mdlref_counter_paramargs_custom ( arg__input, * arg_output )
```

The function prototype in the generated code differs from the preview. The prototype in the generated code (with the additional model parameter arguments) is:

```
sldemo_mdlref_counter_paramargs_custom(
   real_T arg__input,
   real_T *arg_output,
   real_T rtp_lower_saturation_limit,
   real_T rtp_upper_saturation_limit)
```

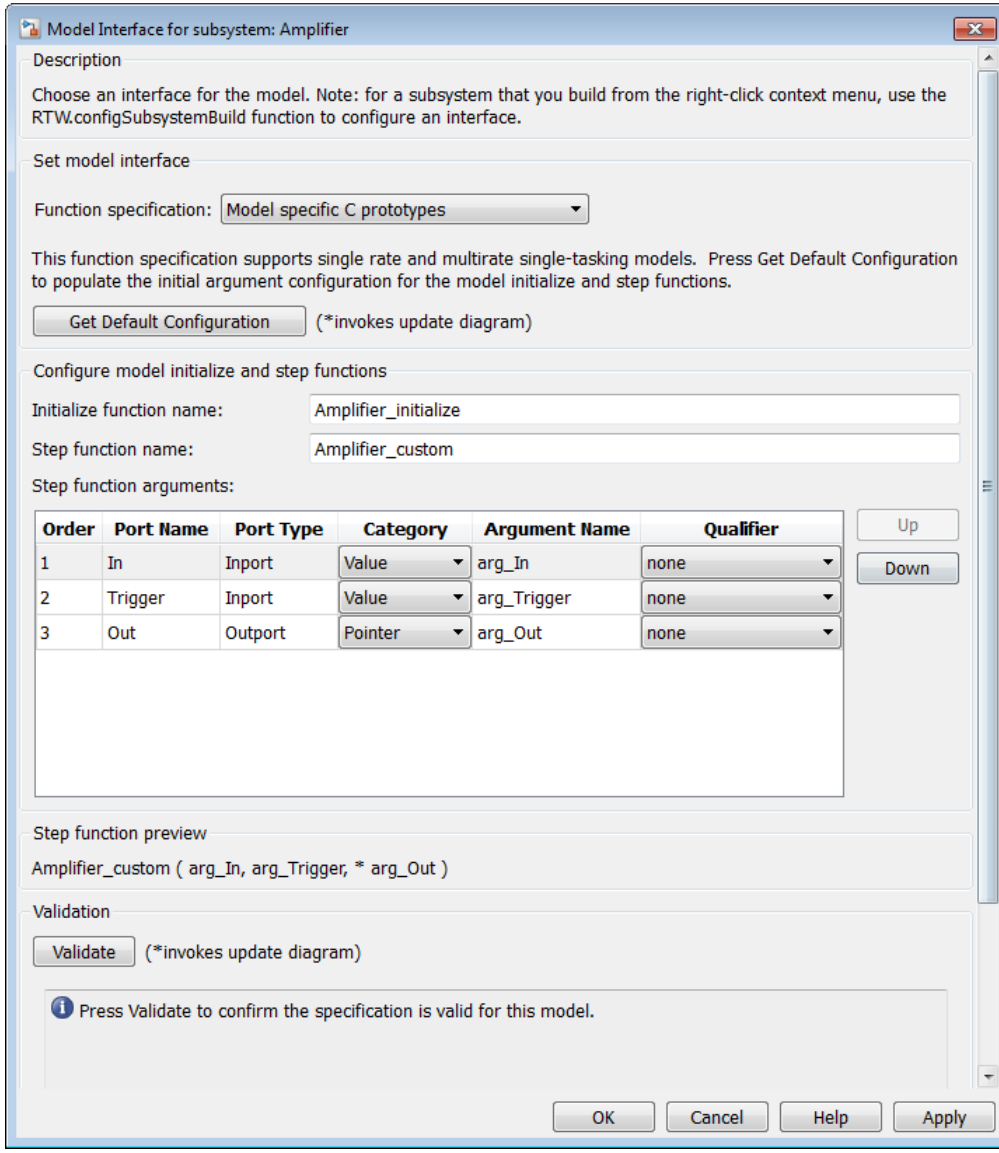
Configure Function Prototypes for Nonvirtual Subsystems

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

- Triggered subsystems
- Enabled subsystems
- Enabled trigger subsystems
- While subsystems
- For subsystems
- Stateflow blocks
- MATLAB function block

To launch the Model Interface for Subsystem dialog box, open the model containing the subsystem and invoke the `RTW.configSubsystemBuild` function.

The Model Interface dialog box for modifying the model-specific C prototypes for the `rtwdemo_counter/Amplifier` subsystem appears as follows:



Right-click building the subsystem generates the step and initialization functions according to the customizations you make.

Sample Procedure for Configuring Function Prototypes

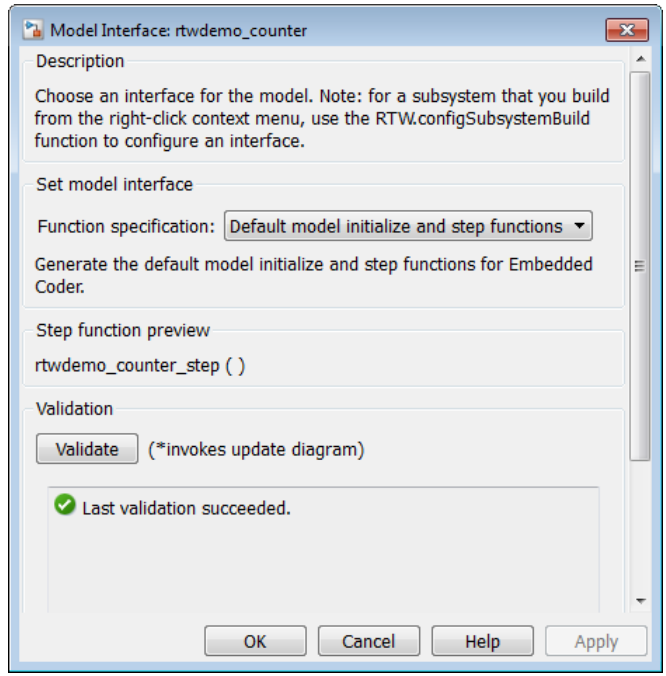
The following procedure shows how to use the **Configure Model Functions** button on the **Code Generation** > **Interface** pane of the Configuration Parameters dialog box to control the model function prototypes when generating code for your Simulink model.

1. Open a MATLAB session and launch the `rtwdemo_counter` model.
2. In the `rtwdemo_counter` Model Editor, double-click the **Generate Code Using Embedded Coder (double-click)** button to generate code for an ERT-based version of `rtwdemo_counter`. The code generation report for `rtwdemo_counter` appears.
3. In the code generation report, click the link for `rtwdemo_counter.c`.
4. In the `rtwdemo_counter.c` code display, locate and examine the generated code for the `rtwdemo_counter_step` and the `rtwdemo_counter_initialize` functions:

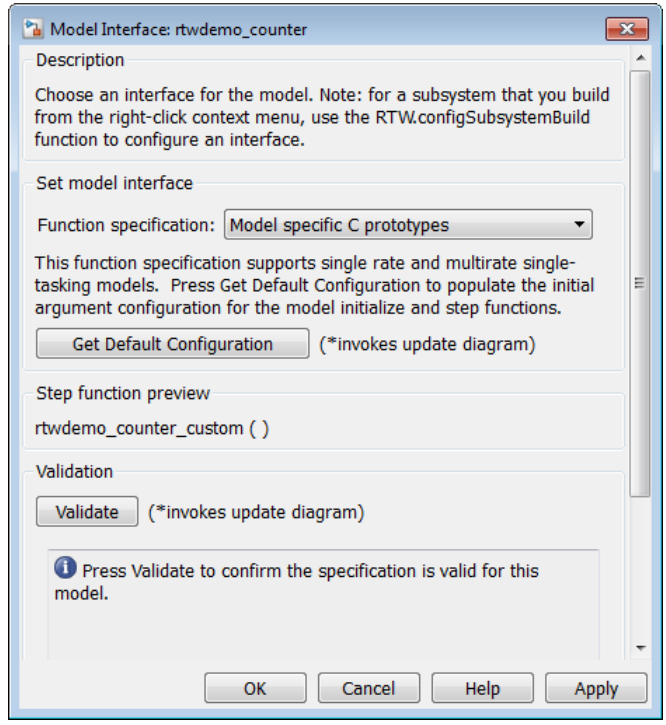
   ```
   /* Model step function */
   void rtwdemo_counter_step(void)
   {
    ...
   }

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

   You can close the report window after you have examined the generated code. Optionally, you can save `rtwdemo_counter.c` and other generated files to a different location for later comparison.
5. From the `rtwdemo_counter` model, open the Configuration Parameters dialog box.
6. Navigate to the **Code Generation** > **Interface** pane and click the **Configure Model Functions** button. The Model Interface dialog box appears.
7. In the initial (`Default model initialize and step functions`) view of the Model Interface dialog box, click the **Validate** button to validate and preview the default function prototype for the `rtwdemo_counter_step` function. The function prototype arguments under **Step function preview** should correspond to the default prototype in step 4.

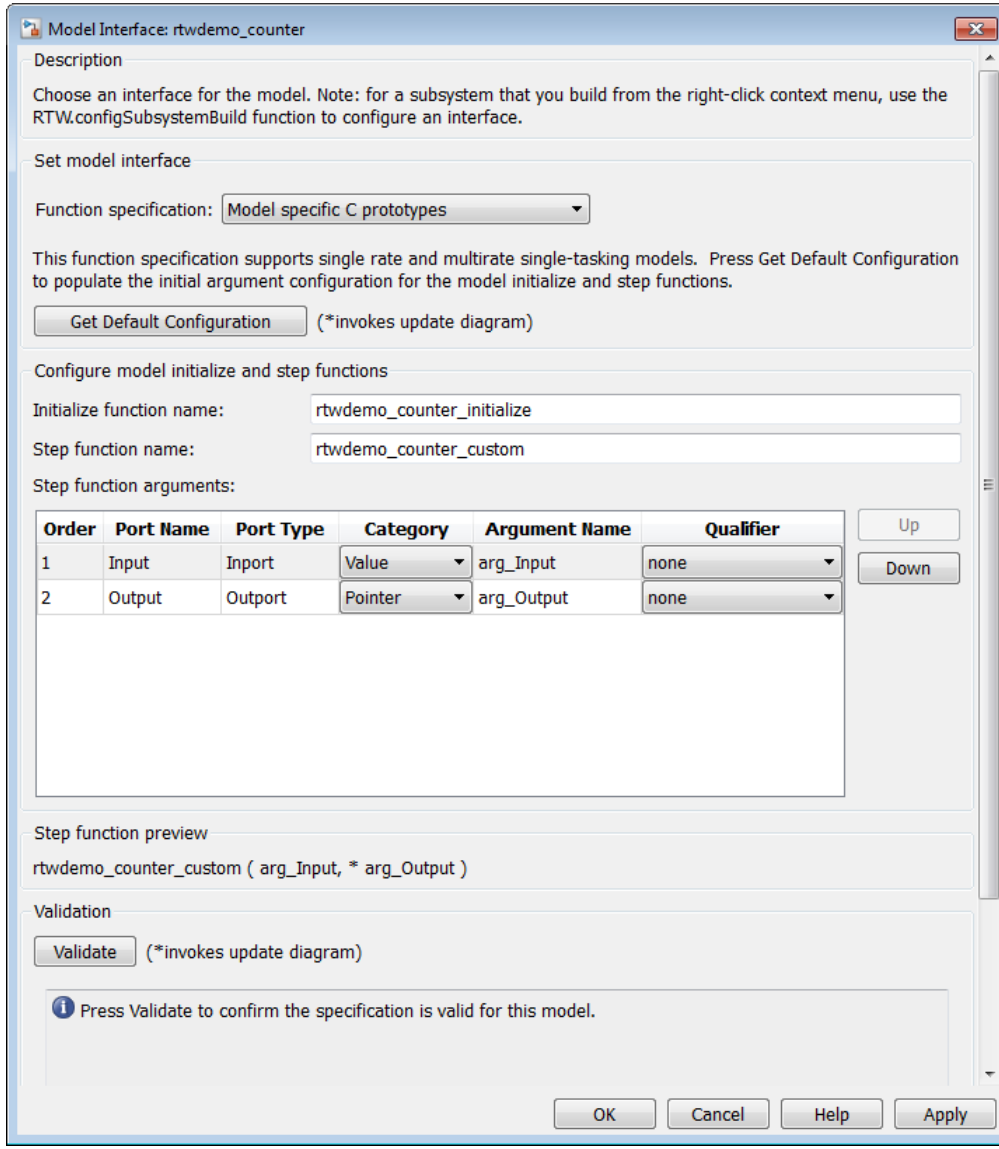
Note: Validation errors in this context prevent successful preview of the default function prototype. Resolve any validation errors to display the preview.



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

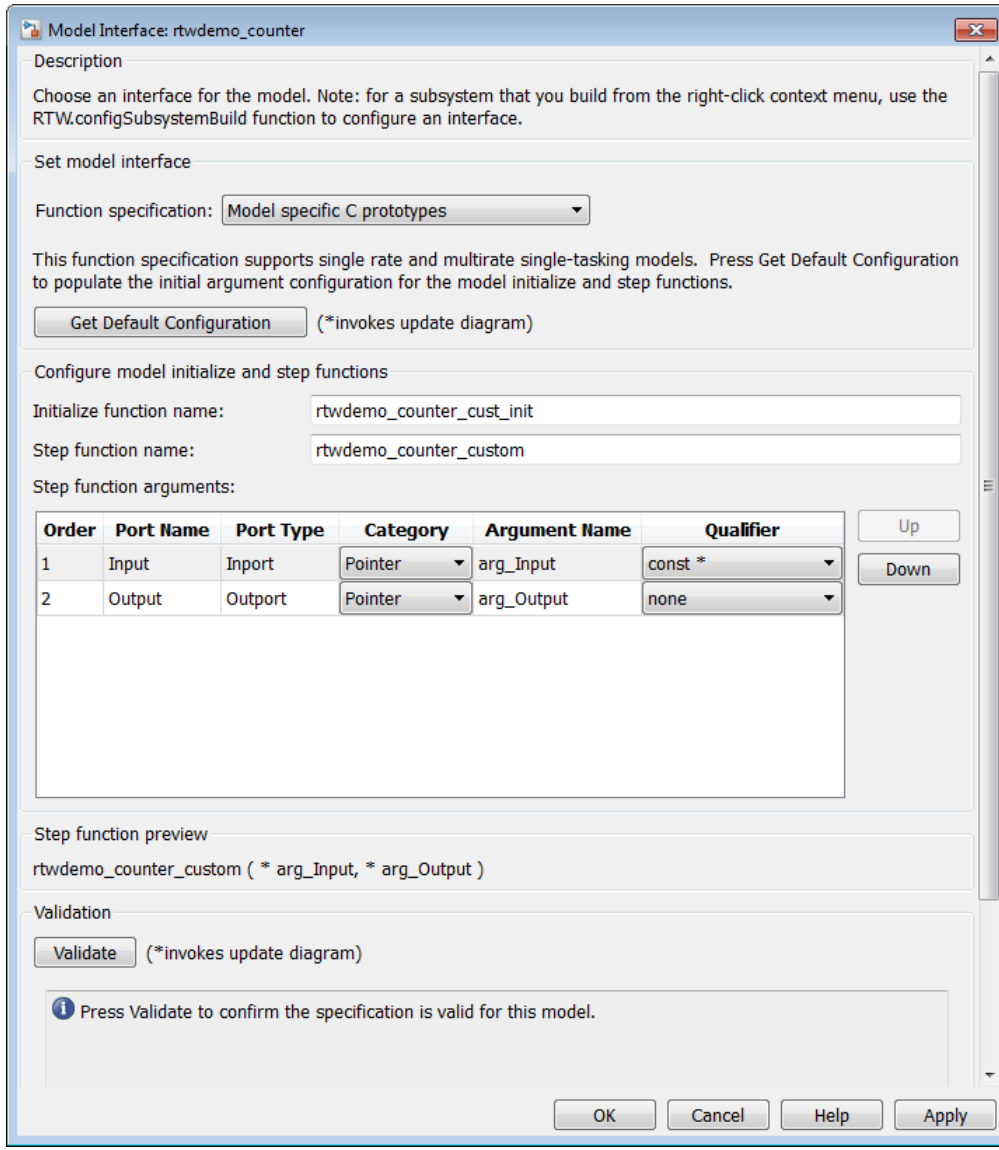


9 In the `Model specific C prototypes` view, click the **Get Default Configuration** button to activate the **Configure model initialize and step functions** subpane.



10 In the **Configure model initialize and step functions** subpane, change **Initialize function name** to `rtwdemo_counter_cust_init`.

11 In the **Configure model initialize and step functions** subpane, in the row for the `Input` argument, change the value of **Category** from `Value` to `Pointer` and change the value of **Qualifier** from `none` to `const *`. The preview reflects your changes.



12 Click the **Validate** button to validate the modified function prototype. The **Validation** subpane displays a message that the validation succeeded.

Note: Validation errors in this context prevent successful code generation. Resolve any validation errors before proceeding. Or, if resolution is not possible, set the **Function specification** field to `Default model initialize and step functions` before proceeding.

---

13 Click **OK** to exit the Model Interface dialog box.

14 Generate code for the model. When the build completes, the code generation report for `rtwdemo_counter` appears.

15 In the code generation report, click the link for `rtwdemo_counter.c`.

16 Locate and examine the generated code for the `rtwdemo_counter_custom` and `rtwdemo_counter_cust_init` functions:

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

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

17 Verify that the generated code is consistent with the function prototype modifications that you specified in the Model Interface dialog box.

Configure Function Prototypes Programmatically

You can use the function prototype control functions (listed in Function Prototype Control Functions), to programmatically control model function prototypes. Typical uses of these functions include:

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

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

  2 Add argument configuration information for your model ports using RTW.ModelSpecificCPrototype.addArgConf.

3. Attach the function prototype to your loaded ERT-based Simulink model using RTW.ModelSpecificCPrototype.attachToModel.
4. Validate the function prototype using RTW.ModelSpecificCPrototype.runValidation.
5. If validation succeeds, save your model and then generate code using the `rtwbuild` function.

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

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

     You can use other function prototype control functions on the returned handle only if the test `isa(obj,'RTW.ModelSpecificCPrototype')` returns 1. If the model does not have a function prototype configuration, the function returns `[]`. If the function returns a handle to an object of type `RTW.FcnDefault`, you cannot modify the existing function prototype.
  2. Use the `Get` and `Set` functions listed in Function Prototype Control Functions to test and reset such items as the function names, argument names, argument positions, argument categories, and argument type qualifiers.
  3. Validate the function prototype using RTW.ModelSpecificCPrototype.runValidation.
  4. If validation succeeds, save your model and then generate code using the `rtwbuild` function.

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

  1. Create a model-specific C function prototype using `obj = RTW.ModelSpecificCPrototype`, where `obj` returns a handle to a newly created, empty function prototype.
  2. Attach the function prototype to your loaded ERT-based Simulink model using RTW.ModelSpecificCPrototype.attachToModel.
  3. Get default configuration information from your model using RTW.ModelSpecificCPrototype.getDefaultConf.

4. Use the `Get` and `Set` functions listed in Function Prototype Control Functions to test and reset such items as the function names, argument names, argument positions, argument categories, and argument type qualifiers.
5. Validate the function prototype using RTW.ModelSpecificCPrototype.runValidation.
6. If validation succeeds, save your model and then generate code using the `rtwbuild` function.

- **Reset the model function prototype to the default ERT function configuration** Create an object of the ERT default function signature. Reset the model function prototype and undo any custom settings, by calling the `RTW.FcnDefault` method, `attachToModel`, as follows:

  ```
  obj = RTW.FcnDefault;
  obj.attachToModel(model);
  ```
  `model` must be a loaded ERT-based model.

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

Function Prototype Control Functions

Function	Description
RTW.ModelSpecificCPrototype.addArgConf	Add step function argument configuration information for Simulink model port to model-specific C function prototype
RTW.ModelSpecificCPrototype.attachToModel	Attach model-specific C function prototype to loaded ERT-based Simulink model
RTW.ModelSpecificCPrototype.getArgCategory	Get step function argument category for Simulink model port from model-specific C function prototype
RTW.ModelSpecificCPrototype.getArgName	Get step function argument name for Simulink model port from model-specific C function prototype

Function	Description
RTW.ModelSpecificCPrototype.getArgPosition	Get step function argument position for Simulink model port from model-specific C function prototype
RTW.ModelSpecificCPrototype.getArgQualifier	Get step function argument type qualifier for Simulink model port from model-specific C function prototype
RTW.ModelSpecificCPrototype.getDefaultConf	Get default configuration information for model-specific C function prototype from Simulink model to which it is attached
RTW.ModelSpecificCPrototype.getFunctionName	Get function names from model-specific C function prototype
RTW.ModelSpecificCPrototype.getNumArgs	Get number of step function arguments from model-specific C function prototype
RTW.ModelSpecificCPrototype.getPreview	Get model-specific C function prototype code previews
`RTW.configSubsystemBuild`	Launch GUI to configure C function prototype or C++ class interface for right-click build of specified subsystem
`RTW.getFunctionSpecification`	Get handle to model-specific C function prototype object
RTW.ModelSpecificCPrototype.runValidation	Validate model-specific C function prototype against Simulink model to which it is attached
RTW.ModelSpecificCPrototype.setArgCategory	Set step function argument category for Simulink model port in model-specific C function prototype
RTW.ModelSpecificCPrototype.setArgName	Set step function argument name for Simulink model port in model-specific C function prototype
RTW.ModelSpecificCPrototype.setArgPosition	Set step function argument position for Simulink model port in model-specific C function prototype

Function	Description
RTW.ModelSpecificCPrototype.setArgQualifier	Set step function argument type qualifier for Simulink model port in model-specific C function prototype
RTW.ModelSpecificCPrototype.setFunctionName	Set function names in model-specific C function prototype

Sample Script for Configuring Function Prototypes

The following sample MATLAB script configures the model function prototypes for the `rtwdemo_counter` model, using the Function Prototype Control Functions.

```
%% Open the rtwdemo_counter model
rtwdemo_counter

%% Select ert.tlc as the System Target File for the model
set_param(gcs,'SystemTargetFile','ert.tlc')

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

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

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

%% Rename the initialization function
setFunctionName(a,'InitFunction','init')

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

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

%% if validation succeeded, generate code and build
if status
    rtwbuild(gcs)
end
```

Verify Generated Code for Customized Functions

You can use software-in-the-loop (SIL) testing to verify the generated code for your customized step and initialization functions. This involves creating a SIL block with your generated code, which then can be integrated into a Simulink model to verify that the generated code provides the same result as the original model or nonvirtual subsystem. For more information, see "Choose a SIL or PIL Approach" on page 64-11.

Function Prototype Control Limitations

The following limitations apply to controlling model function prototypes:

- Function prototype control supports only step and initialization functions generated from a Simulink model.
- Function prototype control supports only single-instance implementations. For standalone targets, you must set **Code interface packaging** to `Nonreusable function` (on the **Code Generation** > **Interface** pane of the Configuration Parameters dialog box). For model reference targets, you must select `One` for the **Total number of instances allowed per top model** parameter (on the **Model Referencing** pane of the Configuration Parameters dialog box).
- For model reference targets, if **Code interface packaging** is set to `Reusable function`, the code generator ignores the setting.
- You must select the **Single output/update function** parameter (on the **All Parameters** tab of the Configuration Parameters dialog box).
- Function prototype control does not support multitasking models. Multirate models are supported, but you must configure the models for single-tasking.
- You must configure root-level inports and outports to use `Auto` storage classes.
- Do not control function prototypes with the static `ert_main.c` provided by MathWorks. Specifying a function prototype control configuration other than the default creates a mismatch between the generated code and the default static `ert_main.c`.
- The code generator removes the data structure for the root inports of the model unless a subsystem implemented by a nonreusable function uses the value of one or more of the inports.
- The code generator removes the data structure for the root outports of the model except when you enable MAT-file logging, or if the sample time of one or more of the outports is not the fundamental base rate (including a constant rate).

- If you copy a subsystem block and paste it to create a new block in either a new model or the same model, the function prototype control interface information from the original subsystem block does not copy to the new subsystem block.
- If you have a Stateflow license, for a Stateflow chart that uses a model root inport value, or that calls a subsystem that uses a model root inport value, you must do one of the following to generate code:
  - Clear the **Execute (enter) Chart At Initialization** check box in the Stateflow chart.
  - Make the Stateflow function a nonreusable function.
  - Insert a Simulink Signal Conversion block immediately after the root inport. In the Signal Conversion block parameters dialog box, select **Exclude this block from 'Block reduction' optimization**.
- If a model root inport value connects to a Simscape conversion block, you must insert a Simulink Signal Conversion block between the root inport and the Simscape conversion block. In the Signal Conversion block parameters dialog box, select **Exclude this block from 'Block reduction' optimization**.
- When building a referenced model that is configured for function prototype control, do not use virtual buses as inputs or outputs to the referenced model. When bus signals cross referenced model boundaries, use nonvirtual buses.
- If the C function prototype control is not the default, the value is ignored for the **Configuration Parameters > Model Referencing > Pass fixed-size scalar root inputs by value for code generation** parameter. For more information, see "Pass fixed-size scalar root inputs by value for code generation" (Simulink).

Related Examples

- "Combine I/O Arguments in Model Step Interface" on page 26-53
- "Customize Prototypes of Step and Initialize Functions Generated for a Model"

Control Generation of C++ Class Interfaces

Using the **Code interface packaging** (Simulink Coder) option `C++ class`, on the **Code Generation** > **Interface** pane of the Configuration Parameters dialog box, you can generate a C++ class interface to model code. The generated interface encapsulates required model data into C++ class attributes and model entry point functions into C++ class methods. The benefits of C++ class encapsulation include:

- Greater control over access to model data
- Ability to multiply instantiate model classes
- Easier integration of model code into C++ programming environments

C++ class encapsulation also works for right-click builds of nonvirtual subsystems. (For information on requirements that apply, see "Configure C++ Class Interfaces for Nonvirtual Subsystems" on page 26-44.)

The general procedure for generating C++ class interfaces to model code is as follows:

1. Configure your model to use an `ert.tlc` system target file provided by MathWorks.
2. Select the `C++` language for your model.
3. Select `C++ class` code interface packaging for your model.
4. Optionally, configure related C++ class interface settings for your model code, using either a graphical user interface (GUI) or application programming interface (API).
5. Generate model code and examine the results.

To get started with an example, see "Simple Use of C++ Class Control" on page 26-24. For more details about configuring C++ class interfaces for your model code, see "Customize C++ Class Interfaces Using Graphical Interfaces" on page 26-31 and "Customize C++ Class Interfaces Programmatically" on page 26-45. For limitations that apply, see "C++ Class Interface Control Limitations" on page 26-50.

Note: For an example of C++ class code generation, see the example model `rtwdemo_cppclass`.

In this section...

In this section...

Simple Use of C++ Class Control

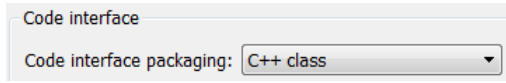
This example illustrates a simple use of `C++ class` code interface packaging. It generates C+ class code interfaces from an example model, without extensive modifications to default settings.

Note: For details about setting C++ class parameters, see the sections that follow this example, beginning with “Customize C++ Class Interfaces Using Graphical Interfaces” on page 26-31.

To generate C++ class interfaces for a Simulink model:

1. Open a model for which you would like to generate C++ class code interfaces. This example uses the model `rtwdemo_counter`.
2. Configure the model to use an `ert.tlc` system target file provided by MathWorks. For example, open the Configuration Parameters dialog box, go to the **Code Generation** pane, select a target value from the **System target file** menu, and click **Apply**.
3. On the **Code Generation** pane of the Configuration Parameters dialog box, set the **Language** parameter to `C++`.

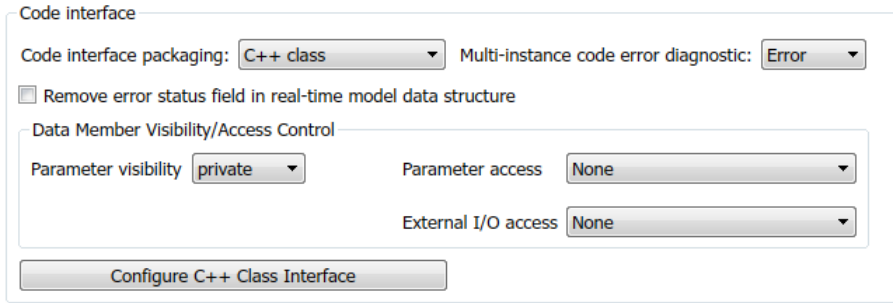
   On the **Code Generation** > **Interface** pane, check that the **Code interface packaging** parameter is set to `C++ class`.

   Click **Apply**.

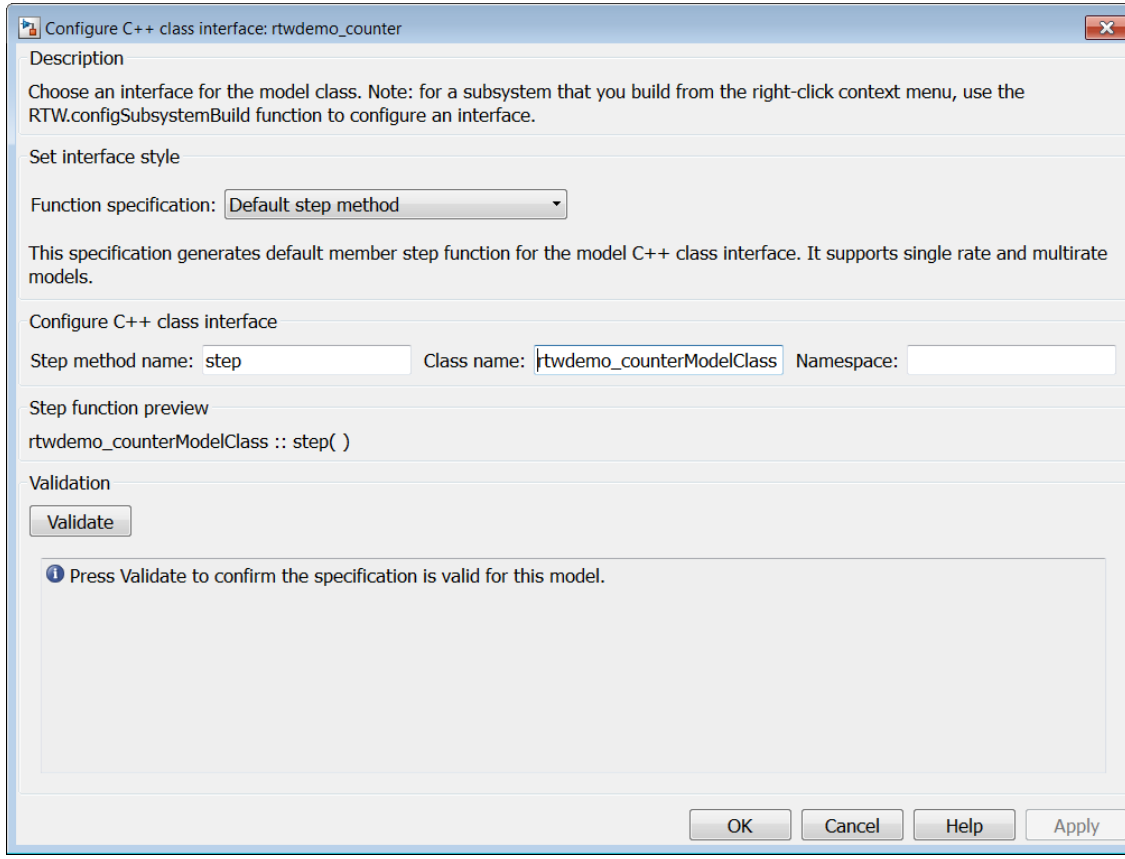
Note: To immediately generate the default style of C++ class code, without exploring the related model configuration options, skip steps 4–8 and go directly to step 9.

4 Go to the **Interface** pane of the Configuration Parameters dialog box and examine the **Code interface** subpane.



When you select `C++ class` code interface packaging for your model, additional C++ class interface controls become available in the **Code interface** subpane. See "Configure Code Interface Options" on page 26-31 for descriptions of these controls. You might want to modify the default settings according to your application.

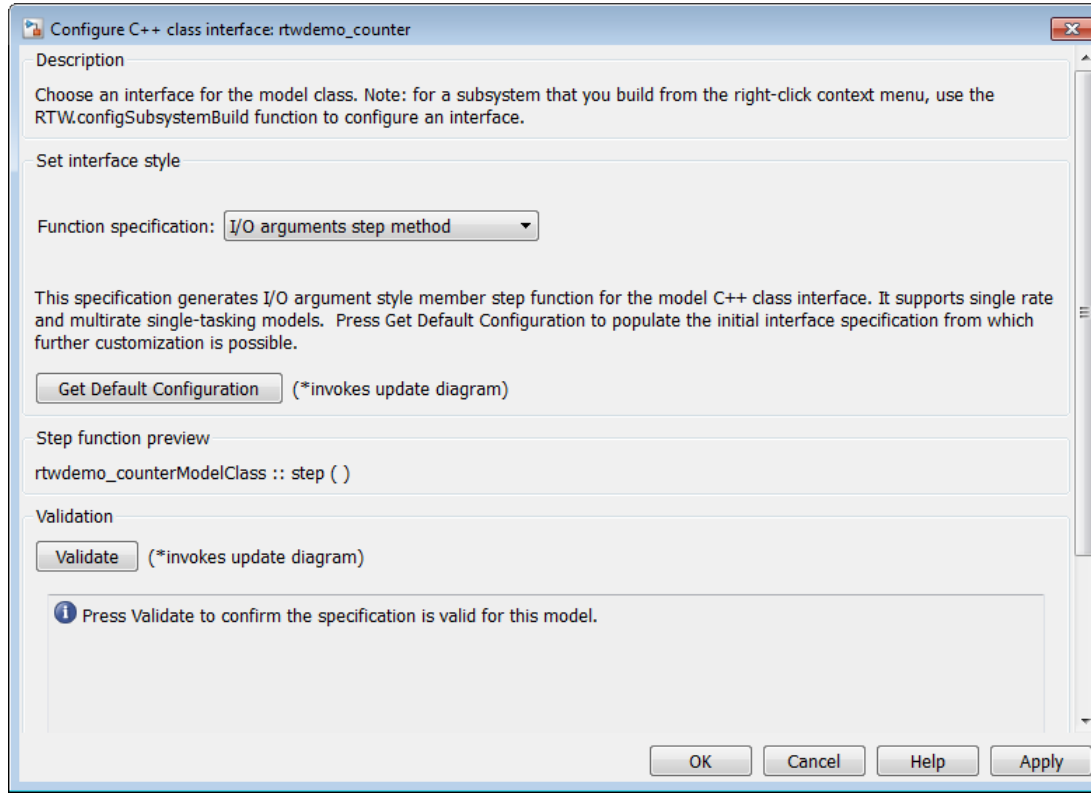
5 Click the **Configure C++ Class Interface** button. This action opens the Configure C++ class interface dialog box, which allows you to configure the step method for your generated model class. The dialog box initially displays a view for configuring a **Default step method** for the model class. In this view, you can specify the model class name, step method name, and namespace for your model.



See “Configure Step Method for Your Model Class” on page 26-34 for descriptions of these controls.

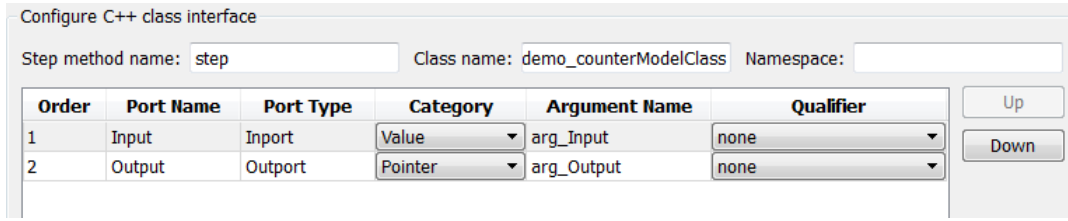
Note: If the default interface style meets your needs, you can skip steps 6–8 and go directly to step 9.

6 If you want root-level model input and output to be arguments on the step method, select the value `I/O arguments step method` from the **Function specification** menu. The dialog box displays a view for configuring an I/O arguments style step method for the model class.



See "Configure Step Method for Your Model Class" on page 26-34 for descriptions of these controls.

7 Click the **Get Default Configuration** button. This action causes a **Configure C ++ class interface** subpane to appear in the dialog box. The subpane displays the initial interface configuration for your model, which provides a starting point for further customization.



See "Passing I/O Arguments" on page 26-37 for descriptions of these controls.

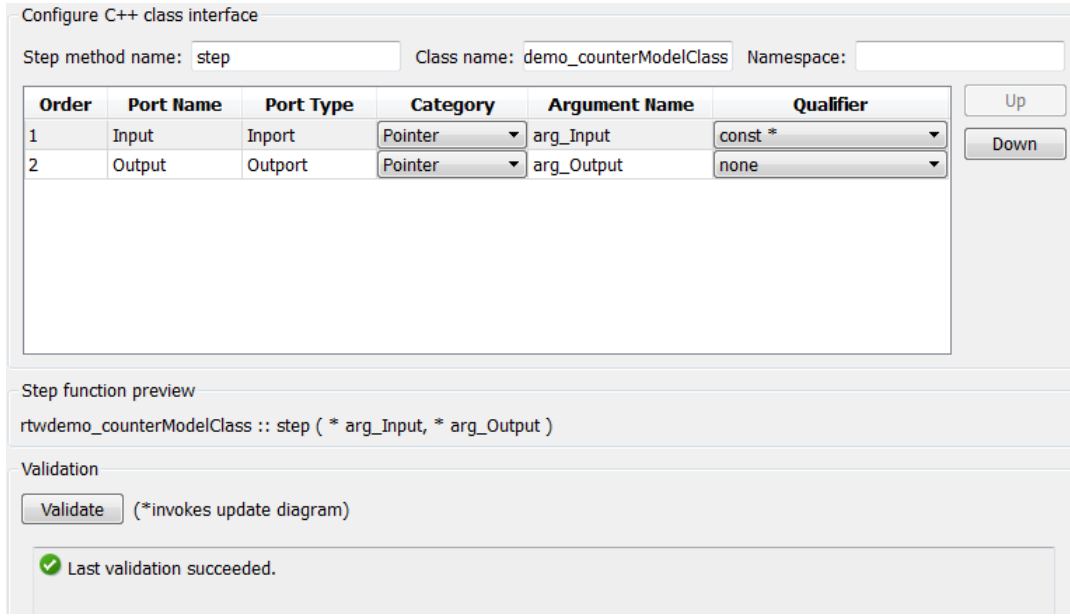
8 Perform this optional step only if you want to customize the configuration of the I/O arguments generated for your model step method.

Note: If you choose to skip this step, you should click **Cancel** to exit the dialog box.

If you choose to perform this step, first you must check that the required option **Remove root level I/O zero initialization** is selected on the **Optimization** pane, and then navigate back to the `I/O arguments step method` view of the Configure C++ class interface dialog box.

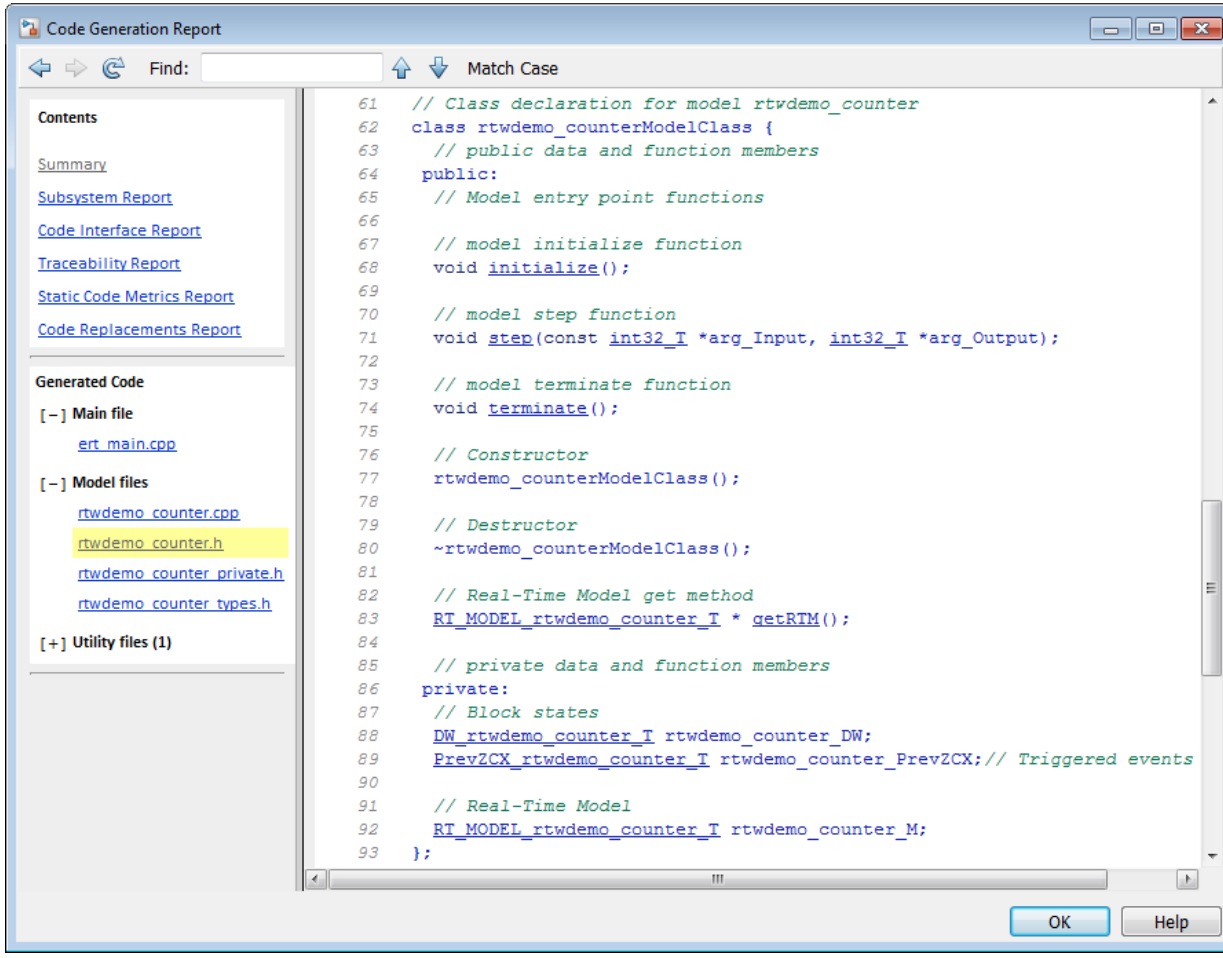
Now you can use the dialog box controls to configure I/O argument attributes. For example, in the **Configure C++ class interface** subpane, in the row for the `Input` argument, you can change the value of **Category** from `Value` to `Pointer` and change the value of **Qualifier** from `none` to `const *`. The preview updates to reflect your changes. Click the **Validate** button to validate the modified interface configuration.

Continue modifying and validating until you are satisfied with the step method configuration.



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

9 Generate code for the model. When the build completes, the code generation report for `rtwdemo_counter` appears. Examine the report and observe that required model data is encapsulated into C++ class attributes and model entry point functions are encapsulated into C++ class methods. For example, click the link for `rtwdemo_counter.h` to see the class declaration for the model.

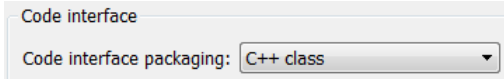


Note: If you configured custom I/O arguments for the model step method (optional step 8), examine the generated code for the step method in `rtwdemo_counter.h` and `rtwdemo_counter.cpp`. The arguments should reflect your changes. For example, if you performed the `Input` argument modifications in step 8, the input argument should appear as `const int32_T *arg_Input`.

Customize C++ Class Interfaces Using Graphical Interfaces

Select C++ Class Code Interface Packaging

To select `C++ class` code interface packaging, in the Configuration Parameters dialog box, on the **Code Generation** pane, set the **Language** parameter to `C++`. Then, in the **Code Generation** > **Interface** pane, check that the **Code interface packaging** parameter is set to `C++ class`:

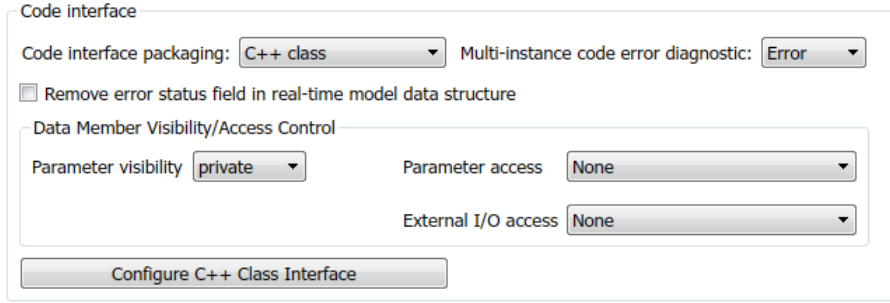


Selecting this value:

- Disables model configuration options that `C++ class` does not support. For details, see "C++ Class Interface Control Limitations" on page 26-50.
- Adds additional C++ class interface parameters, which are described in the next section.

Configure Code Interface Options

When you select `C++ class` code interface packaging for your model, the **Code interface** parameters shown below are displayed on the **Interface** pane.



- **Multi-instance code error diagnostic**

  Specifies the severity level for diagnostics displayed when a model violates requirements for generating multi-instance code.

  - `None` — Proceed with build without displaying a diagnostic message.
  - `Warning` — Proceed with build after displaying a warning message.
  - `Error` (default) — Abort build after displaying an error message.

- **Remove error status field in real-time model data structure**

  Specifies whether to omit the error status field from the generated real-time model data structure `rtModel` (off by default). Selecting this option reduces memory usage.

  Be aware that selecting this option can cause the code generator to omit the `rtModel` data structure from generated code.

- **Parameter visibility**

  Specifies whether to generate the block parameter structure as a `public`, `private`, or `protected` data member of the C++ model class (`private` by default).

- **Parameter access**

  Specifies whether to generate access methods for block parameters for the C++ model class (`None` by default). You can select noninlined access methods (`Method`) or inlined access methods (`Inlined method`).

- **External I/O access**

  Specifies whether to generate access methods for root-level I/O signals for the C++ model class (`None` by default). If you want to generate access methods, you have the following options:

  - Generate either noninlined or inlined access methods.
  - Generate either *per-signal* or *structure-based* access methods. That is, you can generate a series of set and get methods on a per-signal basis, or generate just one set method that takes the address of an external input structure as an argument and, for external outputs (if applicable), just one get method that returns a reference to an external output structure. The generated code for structure-based access methods has the following general form:

    ```
    class ModelClass {
    ...
    ```

```
    // Root inports set method
    void setExternalInputs(const ExternalInputs* pExternalInputs);

    // Root outports get method
    const ExternalOutputs & getExternalOutputs() const;
}
```

Note: This parameter affects generated code only if you are using the default style step method for your model class; *not* if you are explicitly passing arguments for root-level I/O signals using an I/O arguments style step method. For more information, see "Passing Default Arguments" on page 26-35 and "Passing I/O Arguments" on page 26-37.

- **Configure C++ Class Interface**

  Opens the Configure C++ class interface dialog box, which allows you to configure the step method for your model class. For more information, see "Configure Step Method for Your Model Class" on page 26-34.

Interface parameters that are related, but are less commonly used, are displayed in the **All Parameters** tab:

- **Terminate function required**

  Specifies whether to generate the *`model`*`_terminate` method (on by default). This function contains model termination code and should be called as part of system shutdown.

- **Combine signal/state structures**

  Specifies whether to combine global block signals and global state data into one data structure in the generated code (off by default). Selecting this option reduces RAM and improves readability of the generated code.

- **Internal data visibility**

  Specifies whether to generate internal data structures, such as Block I/O, DWork vectors, Runtime model, Zero-crossings, and continuous states, as `public`, `private`, or `protected` data members of the C++ model class (`private` by default).

- **Internal data access**

  Specifies whether to generate access methods for internal data structures, such as Block I/O, DWork vectors, Runtime model, Zero-crossings, and continuous states,

for the C++ model class (`None` by default). You can select noninlined access methods (`Method`) or inlined access methods (`Inlined method`).

- **Generate destructor**

  Specifies whether to generate a destructor for the C++ model class (on by default).

- **Use dynamic memory allocation for model block instantiation** (Simulink Coder)

  For a model containing Model blocks, specifies whether generated code should use dynamic memory allocation, during model object registration, to instantiate objects for referenced models configured with a C++ class interface (off by default). If you select this option, during instantiation of an object for the top model in a model reference hierarchy, the generated code uses the operator `new` to instantiate objects for referenced models.

  Selecting this option frees a parent model from having to maintain information about referenced models beyond its direct children. Clearing this option means that a parent model maintains information about its referenced models, including its direct and indirect children.

Note:

- If you select this option, be aware that a `bad_alloc` exception might be thrown, per the C++ standard, if an out-of-memory error occurs during the use

  of `new`. You must provide code to catch and process the `bad_alloc` exception in case an out-of-memory error occurs for a `new` call during construction of a top model object.

- If **Use dynamic memory allocation for model block instantiation** is selected and the base model contains a Model block, the build process might generate copy constructor and assignment operator functions in the private section of the model class. The purpose of the functions is to prevent pointer members within the model class from being copied by other code. For more information, see "Model Class Copy Constructor and Assignment Operator" on page 26-49.

Configure Step Method for Your Model Class

To configure the step method for your model class, on the **Code Generation > Interface** pane, click the **Configure C++ Class Interface** button, which is available when

you select `C++ class` code interface packaging for your model. This action opens the Configure C++ class interface dialog box, where you can configure the step method for your model class in either of two styles:

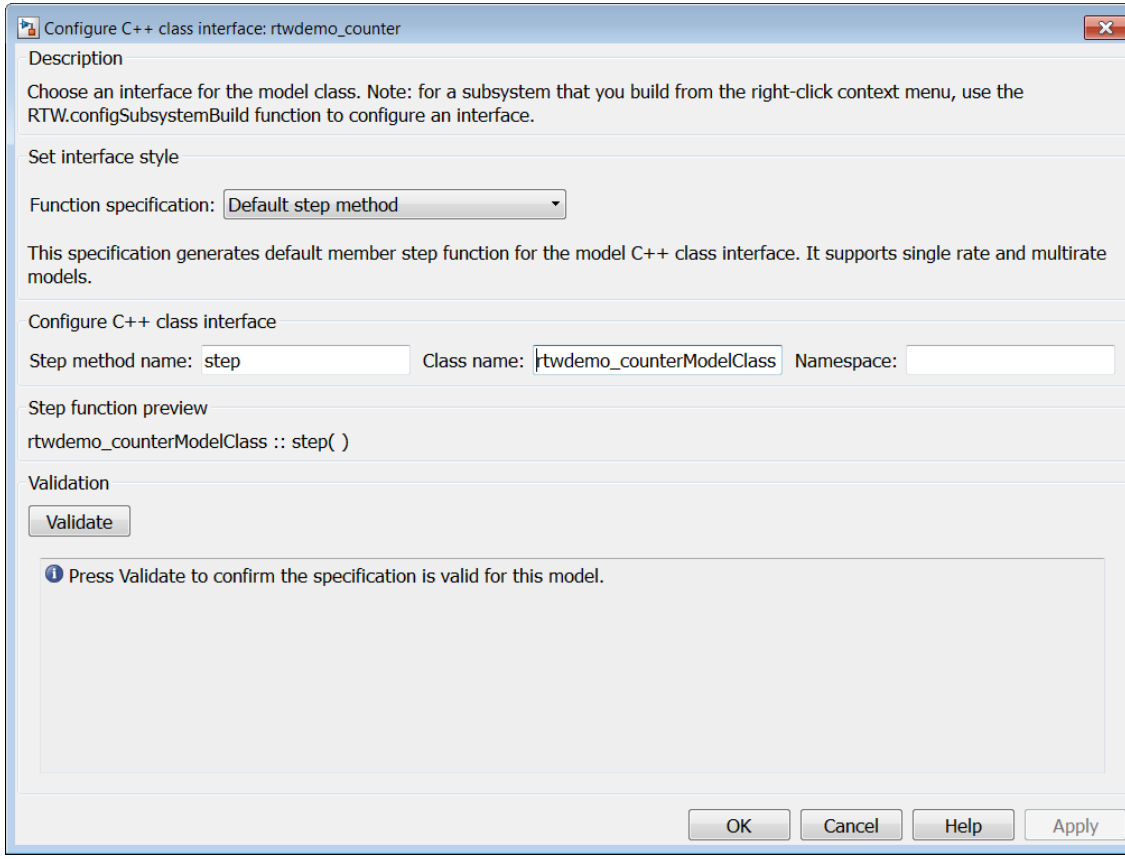
- "Passing Default Arguments" on page 26-35
- "Passing I/O Arguments" on page 26-37

Note: The **Default step method** supports single-rate models and multirate models. The model can be configured for single-tasking operation or multi-tasking operation. This method also supports virtual bus crossing boundaries.

The **I/O arguments step method** supports single-rate models and multirate models. The model can be configured for single-tasking operation.

Passing Default Arguments

The Configure C++ class interface dialog box initially displays a view for configuring a **Default step method** for the model class.



- **Step method name**

  Allows you to specify a step method name other than the default, `step`.

- **Class name**

  Allows you to specify a model class name other than the default, *`model`*`ModelClass`.

- **Namespace**

  Allows you to specify a namespace for the model class. If specified, the namespace is emitted in the generated code for the model class. The **Namespace** parameter provides a means of scoping C++ model classes. In a model reference hierarchy, you can specify a different namespace for each referenced model.

- **Step function preview**

  Displays a preview of the model step function prototype as currently configured. The preview display is dynamically updated after you validate your current configuration.

  Note: The list of step function arguments has an entry for each of the model's root-level I/O ports. This list does not include model parameter arguments that can appear in the generated code when the model is used as a referenced model. For example, a model `sldemo_mdlref_counter_paramargs` has an inport with argument name `arg_input`, an outport with argument name `arg_output`, and a saturation block whose limits have workspace parameter argument names `lower_saturation_limit` and `upper_saturation_limit`.

  The step function preview for this model is:

  ```
  sldemo_mdlref_counter_paramargsModelClass :: step ( arg__input, * arg_output )
  ```

  The function prototype in the generated code differs from the preview. The prototype in the generated code (with the additional model parameter arguments) is:

  ```
  sldemo_mdlref_counter_paramargsModelClass::step (
     real_T arg__input,
     real_T *arg_output,
     real_T rtp_lower_saturation_limit,
     real_T rtp_upper_saturation_limit)
  ```

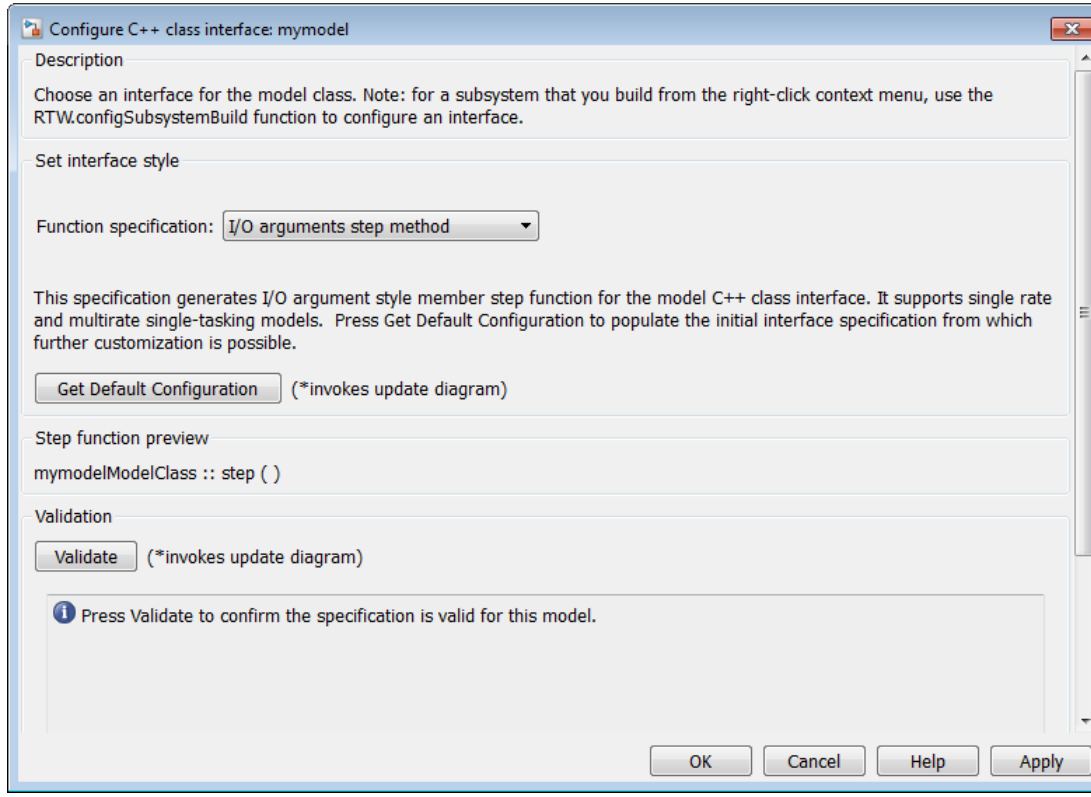
- **Validate**

  Validates your current model step function configuration. The **Validation** pane displays the status and an explanation of any failure.

Passing I/O Arguments

If you select `I/O arguments step method` from the **Function specification** menu, the dialog box displays a view for configuring an I/O arguments style step method for the model class.

Note: To use the I/O arguments style step method, you must select the option **Remove root level I/O zero initialization** on the **Optimization** pane of the Configuration Parameters dialog box.



- **Get Default Configuration**

  Click this button to get the initial interface configuration that provides a starting point for further customization.

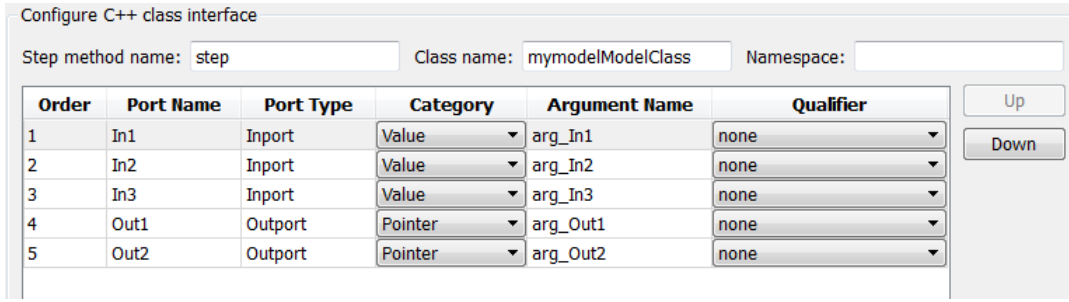
- **Step function preview**

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

- **Validate**

  Validates your current model step function configuration. The **Validation** pane displays the status and an explanation of any failure.

When you click **Get Default Configuration**, the **Configure C++ class interface** subpane appears in the dialog box, displaying the initial interface configuration. For example:



- **Step method name**

  Allows you to specify a step method name other than the default, `step`.

- **Class name**

  Allows you to specify a model class name other than the default, *`model`*`ModelClass`.

- **Namespace**

  Allows you to specify a namespace for the model class. If specified, the namespace is emitted in the generated code for the model class. The **Namespace** parameter provides a means of scoping C++ model classes. In a model reference hierarchy, you can specify a different namespace for each referenced model.

- **Order**

  Displays the numerical position of each argument. Use the **Up** and **Down** buttons to change argument order.

- **Port Name**

  Displays the port name of each argument (not configurable using this dialog box).

- **Port Type**

  Displays the port type, `Inport` or `Outport`, of each argument (not configurable using this dialog box).

- **Category**

Displays the passing mechanism for each argument. To change the passing mechanism for an argument, select `Value`, `Pointer`, or `Reference` from the argument's **Category** menu.

- **Argument Name**

  Displays the name of each argument. To change an argument name, click in the argument's **Argument name** field, position the cursor for text entry, and enter the new name.

- **Qualifier**

  Displays the `const` type qualifier for each argument. To change the qualifier for an argument, select an available value from the argument's **Qualifier** menu. The possible values are:

  - `none`
  - `const` (value)
  - `const*` (value referenced by the pointer)
  - `const*const` (value referenced by the pointer and the pointer itself)
  - `const &` (value referenced by the reference)

Tip: When a model includes a referenced model, the `const` type qualifier for the root input argument of the referenced model's specified step function interface is set to `none` and the qualifier for the source signal in the referenced model's parent is set to a value other than `none`, code generation honors the referenced model's interface specification by generating a type cast that discards the `const` type qualifier from the source signal. To override this behavior, add a `const` type qualifier to the referenced model.

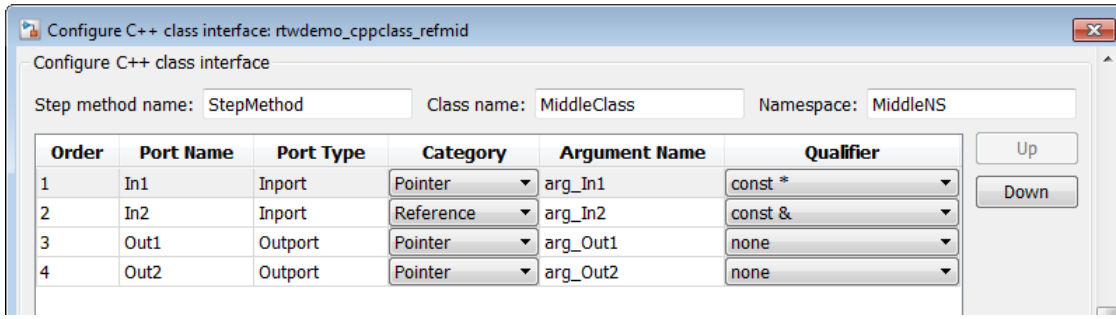
Use Namespaces to Scope C++ Model Classes

Embedded Coder provides namespace control for scoping model classes generated using C++ class code interface packaging. In the Configure C++ class interface dialog box, use the **Namespace** parameter to specify a namespace for a model class. If specified, the namespace is emitted in the generated code for the model class. To scope the C++ model classes in a model reference hierarchy, you can specify a different namespace for each referenced model.

For an example of namespace control, see the example model rtwdemo_cppclass. This model assigns namespaces as follows:

- `TopNS` for top-level model `rtwdemo_cppclass`
- `MiddleNS` for referenced model `rtwdemo_cppclass_refmid`
- `BottomNS` for referenced model `rtwdemo_cppclass_refbot`

If you build the model with its default settings, you can examine the generated header and source files for each model to see where the namespace is emitted. For example, the **Namespace** setting for the model `rtwdemo_cppclass_refmid` is shown below, followed by excerpts of the emitted namespace code in the model header and source files.




```
42   // Class declaration for model rtwdemo_cppclass_refmid
43   namespace MiddleNS {
44     class MiddleClass {
45       // public data and function members
46      public:
47       // Model entry point functions
...
52       // model step function
53       void StepMethod(const real_T *arg_In1, const real_T &arg_In2, real_T
54                       *arg_Out1, real_T *arg_Out2);
...
87     };
88   }
```

```
15   #include "rtwdemo_cppclass_refmid.h"
16   #include "rtwdemo_cppclass_refmid_private.h"
17
18   namespace MiddleNS
19   {
20     // Model step function
21     void MiddleClass::StepMethod(const real_T *arg_In1, const real_T &arg_In2,
22       real_T *arg_Out1, real_T *arg_Out2)
```

```
23      {
...
43      }
...
83    }
```

Combine I/O Arguments in Model Step Interface

When using C function prototype control or C++ class interface control, you can combine a pair of model step function arguments, an input and an output. This merging of input and output allows the code generator to reuse buffers, which can eliminate buffers in the generated code.

The following requirements apply to combining model step function input and output arguments:

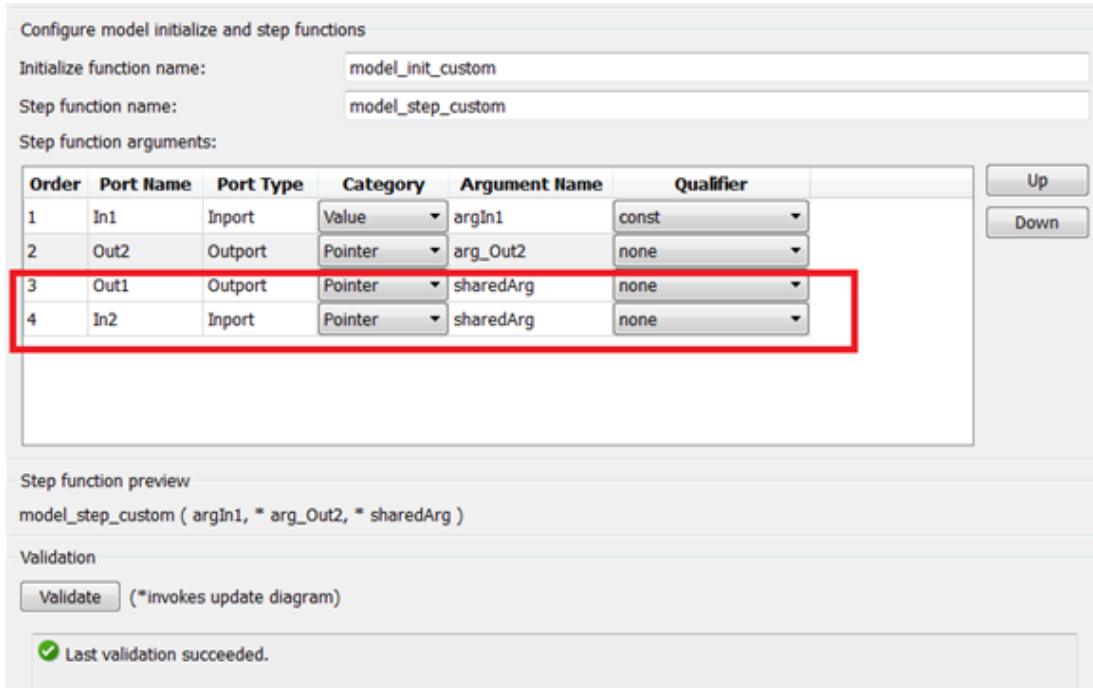
- The input and output arguments must be assigned the same argument name.
- The corresponding inport and outport blocks must have the same data type and sampling rate.

Additionally, the following limitations apply to combining model step function input and output arguments:

- The sample rate of the inport and outport blocks must be the same as the base rate of the model.
- A conditionally executed subsystem cannot drive the outport.
- A single, nonvirtual block output must drive the outport. For example, a Mux block, which merges multiple buffers, cannot drive the outport.

To configure model step function I/O arguments to allow buffer reuse:

1. In the Configuration Parameters dialog box, select the **Code Generation** > **Interface** pane. To initiate C function prototype control, click the **Configure Model Functions** button. To initiate C++ class interface control, click the **Configure C++ Class Interface** button.
2. Navigate to the view that allows you to modify model step function I/O arguments – **Model specific C prototypes** view for C function prototype control or **I/O arguments step method** for C++ class interface control.
3. Select an inport/outport pair, configure their **Category** and **Argument Name** settings to match, and make sure that **Category** is not set to `Value`. Set **Qualifier** to `none` for both ports.



When you generate code from the model, the arguments are combined in the function prototype. For example:

```
35 // Model step function
36 void model_step_custom(const real_T argIn1, boolean_T *arg_Out2, boolean_T
37   *sharedArg)
38 {
```

The shared argument appears in inport read code and outport write code. For example:

```
54
55    *arg_Out2 = !*sharedArg;
56
57    // Update for UnitDelay: '<Root>/Unit Delay' incorporates:
58    //   Update for Inport: '<Root>/In1'
59
60    rtDWork.UnitDelay_DSTATE = argIn1;
61
62    // Logic: '<Root>/LogOp'
63    *sharedArg = (rtb_RelOp1 || rtb_RelOp2);
64  }
```

Configure C++ Class Interfaces for Nonvirtual Subsystems

You can configure C++ class interfaces for right-click builds of nonvirtual subsystems in Simulink models, if the following requirements are met:

- The model is configured for the `C++` language and `C++ class` code interface packaging.
- The subsystem is convertible to a Model block using the function `Simulink.SubSystem.convertToModelReference`. For referenced model conversion requirements, see the Simulink reference page `Simulink.SubSystem.convertToModelReference`.

To configure C++ class interfaces for a subsystem that meets the requirements:

1. Open the containing model and select the subsystem block.
2. Enter the following MATLAB command:

   ```
   RTW.configSubsystemBuild(gcb)
   ```

   where `gcb` is the Simulink function `gcb`, returning the full block path name of the current block.

   This command opens a subsystem equivalent of the Configure C++ class interface dialog sequence that is described in detail in the preceding section, “Configure Step Method for Your Model Class” on page 26-34. (For more information about using the MATLAB command, see `RTW.configSubsystemBuild`.)
3. Use the Configure C++ class interface dialog boxes to configure C++ class settings for the subsystem.
4. Right-click the subsystem and select **C/C++ Code** > **Build This Subsystem**.

5 When the subsystem build completes, you can examine the C++ class interfaces in the generated files and the HTML code generation report.

Customize C++ Class Interfaces Programmatically

If you select the **Code interface packaging** option `C++ class` for your model, you can use the C++ class interface control functions (listed in C++ Class Interface Control Functions) to programmatically configure the step method for your model class.

Typical uses of these functions include:

- **Create and validate a new step method interface, starting with default configuration information from your Simulink model**

  1 Create a model-specific C++ class interface with `obj = RTW.ModelCPPDefaultClass` or `obj = RTW.ModelCPPArgsClass`, where `obj` returns a handle to an newly created, empty C++ class interface.

  2 Attach the C++ class interface to your loaded ERT-based Simulink model using `attachToModel`.

  3 Get default C++ class interface configuration information from your model using `getDefaultConf`.

  4 Use the `Get` and `Set` functions listed in C++ Class Interface Control Functions to test or reset the model class name and model step method name. Additionally, if you are using the I/O arguments style step method, you can test and reset argument names, argument positions, argument categories, and argument type qualifiers.

  5 Validate the C++ class interface using `runValidation`. (If validation fails, use the error message information that`runValidation` returns to address the issues.)

  6 Save your model and then generate code using the `rtwbuild` function.

- **Modify and validate an existing step method interface for a Simulink model**

  1 Get the handle to an existing model-specific C++ class interface that is attached to your loaded ERT-based Simulink model using `obj = RTW.getClassInterfaceSpecification(modelName)`, where `modelName` is a character vector specifying the name of a loaded ERT-based Simulink model, and `obj` returns a handle to a C++ class interface attached to the specified model. If the model does not have an attached C++ class interface configuration, the function returns `[]`.

2. Use the `Get` and `Set` functions listed in C++ Class Interface Control Functions to test or reset the model class name and model step method name. Additionally, if the returned interface uses the I/O arguments style step method, you can test and reset argument names, argument positions, argument categories, and argument type qualifiers.
3. Validate the C++ class interface using `runValidation`. (If validation fails, use the error message information that `runValidation` returns to address the issues.)
4. Save your model and then generate code using the `rtwbuild` function.

Note: You should not use the same model-specific C++ class interface control object across multiple models. If you do, changes that you make to the step method configuration in one model propagate to other models, which is usually not desirable.

C++ Class Interface Control Functions

Function	Description
`attachToModel`	Attach model-specific C++ class interface to loaded ERT-based Simulink model
`getArgCategory`	Get argument category for Simulink model port from model-specific C++ class interface
`getArgName`	Get argument name for Simulink model port from model-specific C++ class interface
`getArgPosition`	Get argument position for Simulink model port from model-specific C++ class interface
`getArgQualifier`	Get argument type qualifier for Simulink model port from model-specific C++ class interface
`getClassName`	Get class name from model-specific C++ class interface
`getDefaultConf`	Get default configuration information for model-specific C++ class interface from Simulink model to which it is attached
`getNamespace`	Get namespace from model-specific C++ class interface
`getNumArgs`	Get number of step method arguments from model-specific C++ class interface
`getStepMethodName`	Get step method name from model-specific C++ class interface

Function	Description
`RTW.configSubsystemBuild`	Open GUI to configure C function prototype or C++ class interface for right-click build of specified subsystem
`RTW.getClass-InterfaceSpecification`	Get handle to model-specific C++ class interface control object
`runValidation`	Validate model-specific C++ class interface against Simulink model to which it is attached
`setArgCategory`	Set argument category for Simulink model port in model-specific C++ class interface
`setArgName`	Set argument name for Simulink model port in model-specific C++ class interface
`setArgPosition`	Set argument position for Simulink model port in model-specific C++ class interface
`setArgQualifier`	Set argument type qualifier for Simulink model port in model-specific C++ class interface
`setClassName`	Set class name in model-specific C++ class interface
`setNamespace`	Set namespace in model-specific C++ class interface
`setStepMethodName`	Set step method name in model-specific C++ class interface

Configure Step Method for Model Class

The following sample MATLAB script configures the step method for the `rtwdemo_counter` model class, using the C++ Class Interface Control Functions.

```
%% Open the rtwdemo_counter model
rtwdemo_counter

%% Select ert.tlc as the System Target File for the model
set_param(gcs,'SystemTargetFile','ert.tlc')

%% Select C++ as the target language for the model
set_param(gcs,'TargetLang','C++')

%% Select C++ class as the code interface packaging for the model
set_param(gcs,'CodeInterfacePackaging','C++ class')

%% Set required option for I/O arguments style step method (cmd off = GUI on)
set_param(gcs,'ZeroExternalMemoryAtStartup','off')

%% Create a C++ class interface using an I/O arguments style step method
a=RTW.ModelCPPArgsClass
```

```
%% Attach the C++ class interface to the model
attachToModel(a,gcs)

%% Get the default C++ class interface configuration from the model
getDefaultConf(a)

%% Move the Output port argument from position 2 to position 1
setArgPosition(a,'Output',1)

%% Reset the model step method name from step to StepMethod
setStepMethodName(a,'StepMethod')

%% Change the Input port argument name, category, and qualifier
setArgName(a,'Input','inputArg')
setArgCategory(a,'Input','Pointer')
setArgQualifier(a,'Input','const *')

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

%% if validation succeeded, generate code and build
if status
    rtwbuild(gcs)
end
```

Specify Custom Storage Class for C++ Class Code Generation

To configure a Simulink parameter, signal, or state to use a custom storage class (CSC) with C++ class code generation:

1. Open an ERT-based model for which **Language** is set to `C++` and **Code interface packaging** is set to `C++ class`.
2. Open the Configuration Parameters dialog box.
3. On the **Code Generation** > **Interface** pane, set the **Multi-instance code error diagnostic** (Simulink Coder) parameter to a value other than `Error`.

4. On the **All Parameters** tab, if the option **Ignore custom storage classes** is selected, clear the option.

   Apply the changes.
5. In the model, select a custom storage class for a parameter, signal, or state. For example, select a signal, open its Properties dialog box, and view its code generation

options. In the **Storage class** drop-down list, select a custom storage class, and then configure its attributes. Apply the changes.

Note: C++ class code generation does not support the following CSCs:

- CSCs with `Volatile` specifications.
- CSCs of type `Other`, except `GetSet`.

---

6 Build the model.

7 In the code generation report, examine the files *`model`*`.h` and *`model`*`.cpp` to observe the use of CSCs in the generated C++ code.

Model Class Copy Constructor and Assignment Operator

Code generation automatically adds a copy constructor and an assignment operator to C++ class declarations when required to securely handle pointer members. The constructor and operator are added as private member functions when both of the following conditions exist:

- The model option **Use dynamic memory allocation for model block instantiation** (Simulink Coder) is set to `on`.
- The base model contains a Model block. The Model block is not directly or indirectly within a subsystem for which **Function packaging** is set to `Reusable function`.

Under these conditions, the software generates a private copy constructor and assignment operator to prevent pointer members within the model class from being copied by other code.

Note: To prevent generation of these functions, consider clearing the option **Use dynamic memory allocation for model block instantiation**.

The code excerpt below shows generated *`model`*`.h` code for a model class that has a pointer member. (Look for instances of `MiddleClass_ptr`). The copy constructor and assignment operator declarations are shown in **bold**.

```
class MiddleClass;    // class forward declaration for <S1>/Bottom model instance
typedef MiddleClass* MiddleClass_ptr;
```

```
...

// Class declaration for model cppclass_top
class Top {
...
  // private data and function members
 private:
  // Block signals
  BlockIO_cppclass_top cppclass_top_B;

  // Block states
  D_Work_cppclass_top cppclass_top_DWork;

  // Real-Time Model
  RT_MODEL_cppclass_top cppclass_top_M;

  // private member function(s) for subsystem '<Root>/Subsystem'
  void cppclass_top_Subsystem_Init();
  void cppclass_top_Subsystem_Start();
  void cppclass_top_Subsystem();

  //Copy Constructor
  Top(const Top &rhs);

  //Assignment Operator
  Top& operator= (const Top &rhs);

  // model instance variable for '<S1>/Bottom model instance'
  MiddleClass_ptr Bottom_model_instanceMDLOBJ1;
};
```

C++ Class Interface Control Limitations

- The `C++ class` code interface packaging option does not support some Simulink model configuration options. Selecting `C++ class` disables the following items in the Configuration Parameters dialog box:
  - **Identifier format control** subpane on the **Symbols** pane
  - **File customization template** parameter on the **Templates** pane

  ---

  Note: The code and data templates on the **Templates** pane are supported for `C++ class` code generation. However, the following template file features that are supported for other language selections are not supported for C++ class generated code:

  - Free-form text outside template sections
  - Custom tokens

    - TLC commands (`<! >` tokens)

  - **Global data placement (custom storage classes only)** subpane on the **Code Placement** pane
  - **Memory Sections** pane
- Among the data exchange interfaces available on the **Interface** pane of the Configuration Parameters dialog box, only the C API interface is supported for `C++ class` code generation. If you select **External mode** or **ASAP2 interface**, code generation fails with a validation error.
- The I/O arguments style of step method specification supports single-rate models and multirate single-tasking models, but not multirate multitasking models.
- If you have a Stateflow license, for a Stateflow chart that resides in a root model configured to use the `I/O arguments step method` function specification, and that uses a model root inport value or calls a subsystem that uses a model root inport value, you must do one of the following to generate code:
  - Clear the **Execute (enter) Chart At Initialization** check box in the Stateflow chart.
  - Insert a Simulink Signal Conversion block immediately after the root inport. In the Signal Conversion block parameters dialog box, select **Exclude this block from 'Block reduction' optimization**.
- If a model root inport value connects to a Simscape conversion block, you must insert a Simulink Signal Conversion block between the root inport and the Simscape conversion block. In the Signal Conversion block parameters dialog box, select **Exclude this block from 'Block reduction' optimization**.
- When building a referenced model that is configured to generate a C++ class interface:
  - Do not use a C++ class interface in cases when a referenced model cannot have a combined output/update function. Cases include a model that has a continuous sample time or saves states.
  - Do not use virtual buses as inputs or outputs to the referenced model when the referenced model uses the I/O arguments step method. When bus signals cross referenced model boundaries, either use nonvirtual buses or use the Default step method.
- If the C++ encapsulation interface is not the default, the value is ignored for the **Configuration Parameters** > **Model Referencing** > **Pass fixed-size scalar root**

inputs by value for code generation parameter. For more information, see "Pass fixed-size scalar root inputs by value for code generation" (Simulink).

Related Examples

- "Combine I/O Arguments in Model Step Interface" on page 26-53
- "Customize C++ Encapsulation Interface to Generated Code"

Combine I/O Arguments in Model Step Interface

When using C function prototype control or C++ class interface control, you can combine a pair of model step function arguments, an input and an output. This merging of input and output allows the code generator to reuse buffers, which can eliminate buffers in the generated code.

The following requirements apply to combining model step function input and output arguments:

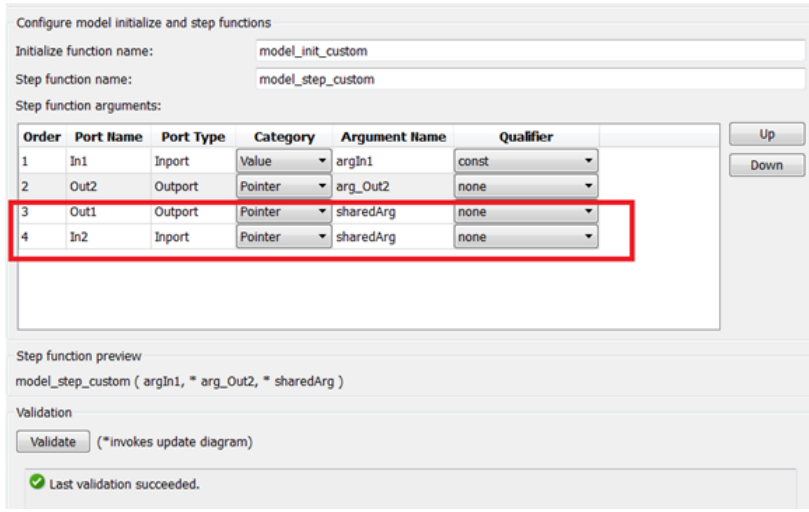
- The input and output arguments must be assigned the same argument name.
- The corresponding inport and outport blocks must have the same data type and sampling rate.

Additionally, the following limitations apply to combining model step function input and output arguments:

- The sample rate of the inport and outport blocks must be the same as the base rate of the model.
- A conditionally executed subsystem cannot drive the outport.
- A single, nonvirtual block output must drive the outport. For example, a Mux block, which merges multiple buffers, cannot drive the outport.

To configure model step function I/O arguments to allow buffer reuse:

1. In the Configuration Parameters dialog box, select the **Code Generation** > **Interface** pane. To initiate C function prototype control, click the **Configure Model Functions** button. To initiate C++ class interface control, click the **Configure C++ Class Interface** button.
2. Navigate to the view that allows you to modify model step function I/O arguments – **Model specific C prototypes** view for C function prototype control or **I/O arguments step method** for C++ class interface control.
3. Select an inport/outport pair, configure their **Category** and **Argument Name** settings to match, and make sure that **Category** is not set to `Value`. Set **Qualifier** to `none` for both ports.



When you generate code from the model, the arguments are combined in the function prototype. For example:

```
// Model step function
void model_step_custom(const real_T argIn1, boolean_T *arg_Out2, boolean_T
  *sharedArg)
{
```

The shared argument appears in inport read code and outport write code. For example:

```

  *arg_Out2 = !*sharedArg;

  // Update for UnitDelay: '<Root>/Unit Delay' incorporates:
  //   Update for Inport: '<Root>/In1'

  rtDWork.UnitDelay_DSTATE = argIn1;

  // Logic: '<Root>/LogOp'
  *sharedArg = (rtb_RelOp1 || rtb_RelOp2);
}
```

Related Examples

- “Customize C++ Encapsulation Interface to Generated Code”

Generate Modular Function Code

The Embedded Coder software provides a Subsystem Parameters dialog box 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.

About Nonvirtual Subsystem Code Generation

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. The subsystem data structure is declared independently from the parent model data structures. A subsystem with separate data has its own block I/O and `DWork` data structure. As a result, the generated code for the subsystem is easier to trace and test. The data separation also tends to reduce the maximum size of global data structures throughout the model, because they are split into multiple data structures.

To use the **Function with separate data** parameter,

- Your model must use an ERT-based system target file (requires a Embedded Coder license).
- Your subsystem must be configured to be atomic or conditionally executed. For more information, see "Systems and Subsystems" (Simulink).
- Your subsystem must use the `Nonreusable function` setting for **Code Generation** > **Function packaging**.

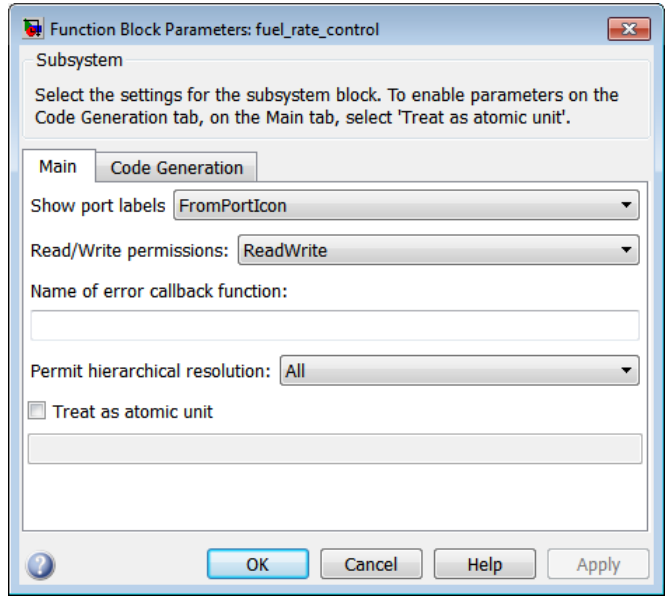
To configure your subsystem for generating modular function code, you invoke the Subsystem Parameters dialog box and make a series of selections to display and enable the **Function with separate data** option. See "Configure Subsystem for Generating Modular Function Code" on page 26-56 and "Modular Function Code for Nonvirtual Subsystems" on page 26-60 for details. For limitations that apply, see "Nonvirtual Subsystem Modular Function Code Limitations" on page 26-66.

For more information about generating code for atomic subsystems, see the sections "Code Generation of Subsystems" (Simulink Coder) and "Generate Code and Executables for Individual Subsystem" (Simulink Coder).

Configure Subsystem for Generating Modular Function Code

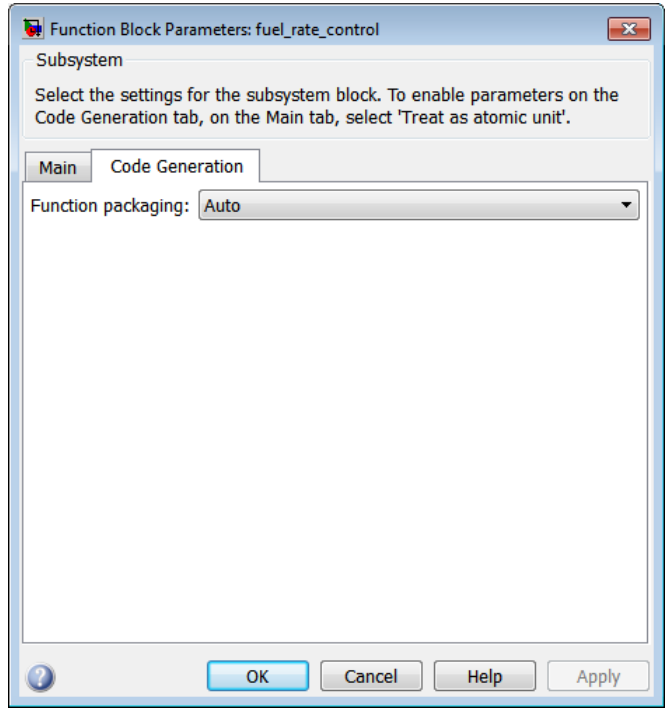
This section summarizes the steps to configure a nonvirtual subsystem in a Simulink model for modular function code generation.

1. Verify that the Simulink model containing the subsystem uses an ERT-based system target file (see the **System target file** parameter on the **Code Generation** 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 **Block Parameters (Subsystem)**). The dialog box for an atomic subsystem is shown below. (In the dialog box for a conditionally executed subsystem, the dialog box option **Treat as atomic unit** is greyed out, and you can skip Step 3.)

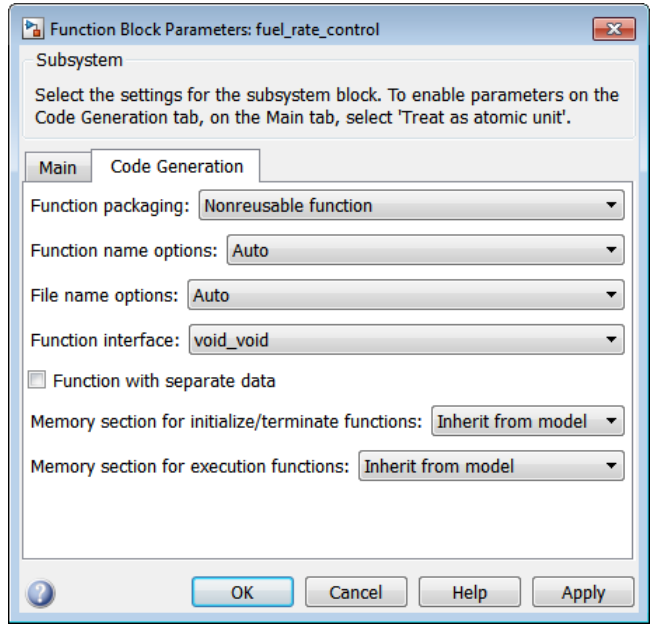


3. If the Subsystem Parameters dialog box option **Treat as atomic unit** is available for selection but not selected, the subsystem is neither atomic nor conditionally

executed. Select the option **Treat as atomic unit**, which enables **Function packaging** on the **Code Generation** tab. Select the **Code Generation** tab.

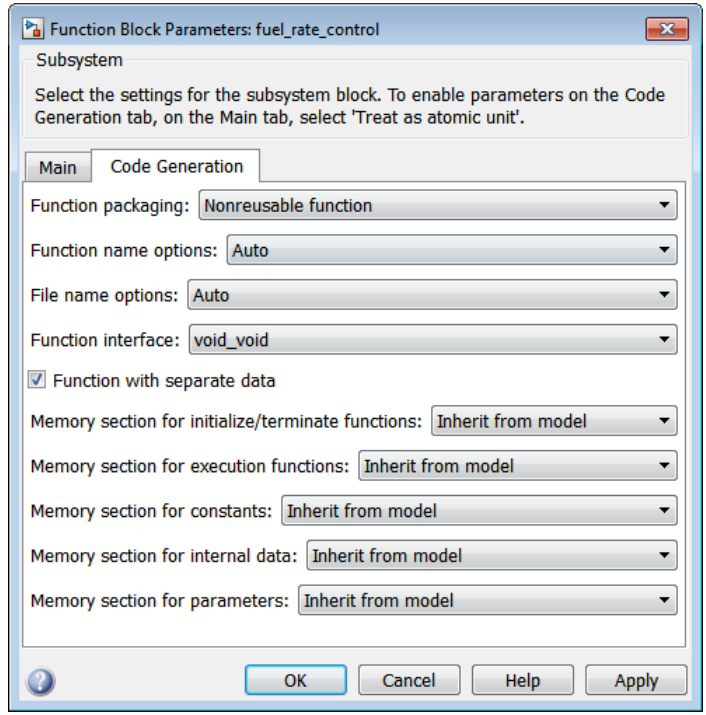


4 For the **Function packaging** parameter, select the value `Nonreusable function`. After you make this selection, the **Function with separate data** option is displayed.



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

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



Note: To control the naming of the subsystem function and the subsystem files in the generated code, you can modify the subsystem parameters **Function name options** and **File name options**.

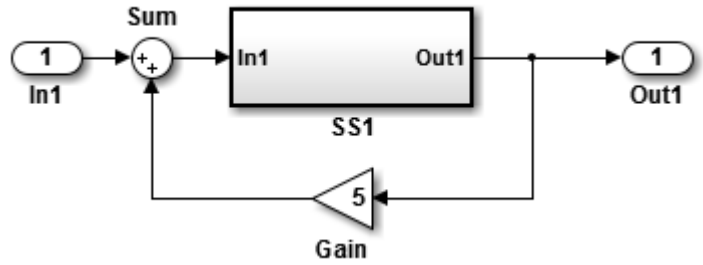
6 To save your subsystem parameter settings and exit the dialog box, click **OK**.

This completes the subsystem configuration for generating 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 “Code Generation of Subsystems” (Simulink Coder). For examples of generated subsystem function code, see “Modular Function Code for Nonvirtual Subsystems” on page 26-60.

Modular Function Code for Nonvirtual Subsystems

To illustrate the selection of 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.

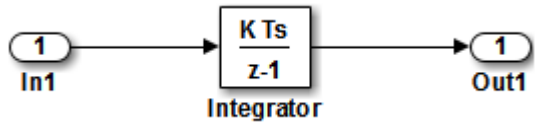
1 Open MATLAB and launch the model `rtwdemo_atomic` using the MATLAB command rtwdemo_atomic.



Examine the Simulink model. This model shows how to preserve the boundary of a virtual subsystem. By selecting the Subsystem Parameters option **Treat as atomic unit**, you guarantee that the code for that subsystem executes as an atomic unit. When a system is marked as atomic, you can specify how the subsystem is represented in code with the Subsystem Parameters option **Code Generation Function Packaging**. You can specify that the subsystem is translated to any of the following types of implementation:

- `Inline`: Inline the subsystem code at the call sites
- `Function`: A void/void function with I/O and internal data in global data structure
- `Reusable Function`: A reentrant function with data passed in as part of function arguments
- `Auto`: Let the code generator optimize the implementation based on context

2 Double-click the SS1 subsystem and examine the contents.



Close the subsystem window when you are finished.

3 Right-click the SS1 subsystem, select Block Parameters (Subsystem) from the context menu, and examine the settings. Simulink and the code generator can avoid "artificial" algebraic loops when the subsystem is made atomic with the subsystem option **Minimize algebraic loop occurrences**.

Close the Block Parameters dialog box when you are finished.

4 Change the **System target file** for the mode from `grt.tlc` to `ert.tlc`. Select the **Configuration Parameters** > **Code Generation** tab and specify `ert.tlc` for the **System target file** parameter. Click **OK** twice to confirm the change. Using the ERT target provides more code generation options for the atomic subsystem.

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

a In the `rtwdemo_atomic` model, right-click the SS1 subsystem and select **Block Parameters (Subsystem)**. In the Subsystem Parameters dialog box that appears, verify that

- On the **Main** tab, **Treat as atomic unit** is selected
- On the **Code Generation** tab, `User specified` is selected for **Function name options**
- On the **Code Generation** tab, `myfun` is specified for **Function name**

b In the Subsystem Parameters dialog box, on the **Code Generation** tab verify that

i `Nonreusable function` is selected for the **Function packaging** parameter. After this selection, additional parameters and options appear.

ii `Use function name` is selected for the **File name options** parameter. This selection is optional but simplifies the later task of code comparison by causing the atomic subsystem function code to be generated into the files `myfun.c` and `myfun.h`.

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

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

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

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

- On the **Main** tab, **Treat as atomic unit** is selected
- On the **Code Generation** tab, `Function` is selected for **Function packaging**
- On the **Code Generation** tab, `User specified` is selected for **Function name options**
- On the **Code Generation** tab, `myfun` is specified for **Function name**
- On the **Code Generation** tab, `Use function name` is specified for **File name options**

b In the Subsystem Parameters dialog box, on the **Code Generation** tab, 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, `rtwdemo_atomic2` in `d:/atomic`.

7 Generate code for each model, `rtwdemo_atomic1` and `rtwdemo_atomic2`.

8 In the generated code directories, compare the *`model`*`.c/.h` and `myfun.c/.h` files generated for the two models. For code comparison discussion, see "H File Differences for Nonvirtual Subsystem Function Data Separation" on page 26-63 and "H File Differences for Nonvirtual Subsystem Function Data Separation" on page 26-63"C File Differences for Nonvirtual Subsystem Function Data Separation" on page 26-64.

In this example, there are not 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` and `rtwdemo_atomic2` help illustrate the selection of the **Function with separate data** option for nonvirtual subsystems.

1. Selecting **Function with separate data** causes `typedef`s 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`, `typedef`s 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` and `rtwdemo_atomic2` illustrate the selection of 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 Embedded Coder license).
- The nonvirtual subsystem to which the option is applied cannot have multiple sample times or continuous sample times; that is, the subsystem must be single-rate with a discrete sample time.
- The nonvirtual subsystem cannot contain continuous states.
- The nonvirtual subsystem cannot output function call signals.
- The nonvirtual subsystem cannot contain noninlined S-functions.
- The generated files for the nonvirtual subsystem will reference model-wide header files, such as *`model`*`.h` and *`model`*`_private.h`.
- The **Function with separate data** option is incompatible with the **Classic call interface** option, located on the **Code Generation** > **Interface** pane of the Configuration Parameters dialog box. Selecting both generates an error.
- The **Function with separate data** option is incompatible with setting **Code interface packaging** to `Reusable function` (**Code Generation** > **Interface** pane). Selecting both generates an error.

More About

- "Design Models for Generated Embedded Code Deployment" on page 1-2

Configure Simulink Function Code Interface

With Embedded Coder, you can customize the generated C/C++ function interfaces for Simulink Function and Function Caller blocks. Function code interface configuration supports easier integration of generated code with functions or function calls in external code, and customizations for coding standards or design requirements.

By opening a dialog box from a selected Simulink Function or Function Caller block, you can customize the C/C++ function prototype generated for that block. Your changes for the selected block also update other corresponding Simulink Function and Function Caller blocks in the model. You can change the generated C/C++ function name, and the names, type qualifiers, and order of function arguments. Your changes do not graphically alter the model and do not affect the Simulink function prototype defined in the block.

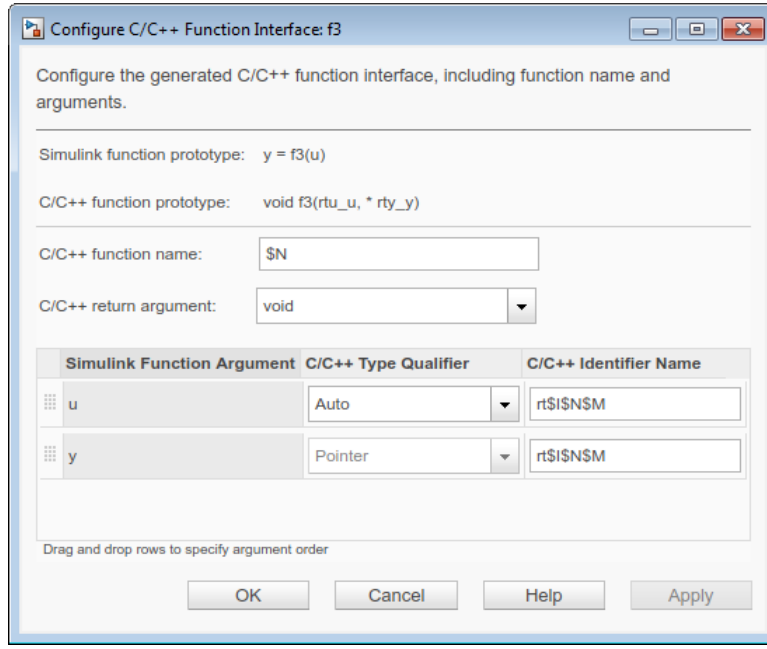
Embedded Coder supports Simulink function code interface configuration for ERT and ERT-derived targets, except for the AUTOSAR target. Function code interface configuration applies to global Simulink functions, and does not apply to private Simulink functions.

Customize Generated C/C++ Function Interface for Simulink Function Block

In this example, you modify the generated C/C++ function interface for a Simulink Function block, and generate C code with the specified changes.

1. Open the example model `rtwdemo_functions`. Save it to a writable work area.
2. Right-click the Simulink Function block `f3`. In the right-click context menu, select **C/C++ Code** > **Configure C/C++ Function Interface**. The Configure C/C++ Function Interface dialog box opens.

   The dialog box displays the Simulink function prototype defined in the block, `y = f3(u)`, and the initial default C/C++ function prototype, `void f3(rtu_u, *rty_y)`.



3 Examine the dialog box settings for the C/C++ function name and the C/C++ argument identifier names — `$N` and `rt$I$N$M`.

In the Configuration Parameters dialog box, **Code Generation** > **Symbols** pane, the **Subsystem method arguments** parameter defines the default identifier format for Simulink Function arguments.

`rt` is a text prefix. `$I`, `$N`, and `$M` are identifier format macros. For more information, see "Subsystem method arguments" (Simulink Coder).

4 In the Configure C/C++ Function Interface dialog box, modify the function and argument identifier names. Changes that you make in the dialog box override model configuration parameter defaults.

In the **C/C++ function name** field, and in the **C/C++ Identifier Name** field for each function argument, enter a custom name or identifier format. Specify valid C-

identifier characters, identifier format macros, or a combination. This example uses function name `function3` and, for both arguments, identifier format `$N$M`.

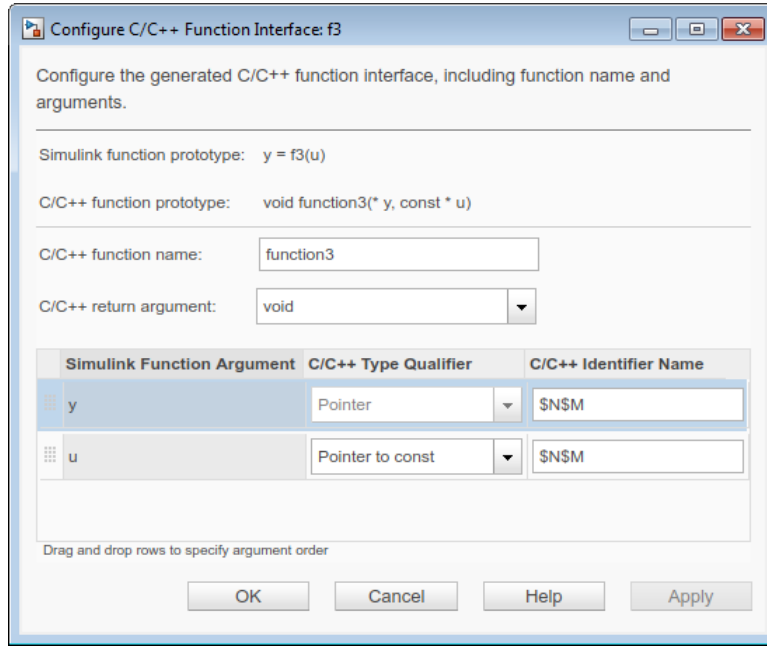
To see tips about available macros, hover over the **C/C++ function name** and **C/C++ Identifier Name** fields. For more information about the identifier format macros, see "Identifier Format Control" on page 36-22.

5 For the `u` argument, change the **C/C++ Type Qualifier** field from `Auto` to `Pointer to const`.

6 To reorder the generated arguments, drag the `y` argument row above the `u` argument row, and drop.

7 Click **Apply** and examine the updated C/C++ function prototype: `void function3(* y, const * u)`.

Your modifications, whether made to a Simulink Function block or a Function Caller block, affect code generation for the Simulink Function block and corresponding Function Caller blocks in the model.



Optionally, you could change **C/C++ return argument** from `void` to `y`. The resulting C/C++ function prototype is `y = function3(const * u)`.

8 Save the model changes and generate code for `rtwdemo_functions`.

9 Open the generated file `rtwdemo_functions.c` and search for `function3`. The generated function code reflects the changes to the generated C/C++ function prototype.

```
/* Output and update for Simulink Function: '<Root>/f3' */
void function3(real_T *y, const real_T *u)
{
...
  adder_h(rtB.Subtract, rtU.U2, *u, &rtB.FunctionCaller);
  ...
  *y = rtB.FunctionCaller;
}
```

Simulink Function Code Interface Limitations

- Simulink function code interface configuration is incompatible with C++ class interface control, in which you configure C++ class interfaces for model entry point functions.
- Simulink function code interface configuration does not support Simulink functions and function callers in Stateflow.

See Also

Function Caller | Simulink Function

More About

- “Customize Generated Identifier Naming Rules” on page 36-15
- “Function and Class Interfaces”
- “Design Models for Generated Embedded Code Deployment” on page 1-2

Memory Sections in Embedded Coder

Control Data and Function Placement in Memory by Inserting Pragmas

In this section...

Some hardware targets run code more efficiently if different kinds of data and functions are stored in specific locations in computer memory. 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 model signals, block parameters, and states. For example, you can specify `const` declarations, or compiler-specific `#pragma` statements for allocation of storage in ROM or flash memory sections.

The Embedded Coder software provides a memory section capability that allows you to insert comments and pragmas and to qualify constants as `volatile` in the generated code for:

- Data in custom storage classes
- Model-level functions
- Model-level internal data
- Subsystem functions
- Subsystem internal data

Pragmas inserted into the 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.

To apply a memory section to groups of data or to model functions such as `initialize` and `step`, you use the Configuration Parameters dialog box. To apply a memory section to individual signals and parameters, you must associate the memory section with a custom storage class. For more information about custom storage classes, see "Introduction to Custom Storage Classes" on page 23-2.

Define Memory Sections

To define memory sections, you must create a data class package using MATLAB class syntax. For more information about creating a data class package, see "Define Data Classes" (Simulink).

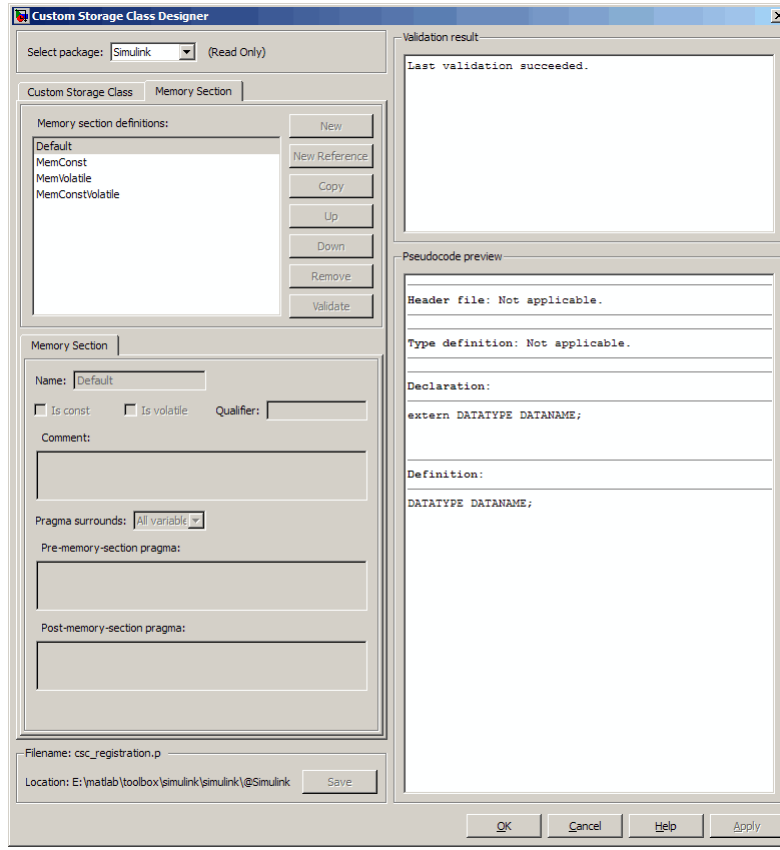
Edit Memory Section Properties

To create new memory sections or specify memory section properties:

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

   The Model Explorer appears.
2. Choose **Tools > Custom Storage Class Designer** in the Model Explorer window.

   A notification box appears that states **Please Wait ... Finding Packages**. After a brief pause, the notification box closes and the Custom Storage Class Designer appears.
3. Click the **Memory Section** tab of the Custom Storage Class Designer. The **Memory Section** pane initially looks like this:



4 If you intend to create or change memory section definitions, use the **Select package** field to select a writable package.

For more detailed information about the controls on the **Memory Section** pane, see "Design Custom Storage Classes and Memory Sections" on page 23-34.

Specify 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.

Specify Comment and Pragma Text

To specify a comment, prepragma, or postpragma for a memory section, enter the comment in the text boxes on the left side of the Custom Storage Class Designer. In the text boxes, you can type multiple lines separated by ordinary Returns.

Surround Individual Definitions with Pragmas

If the **Pragma surrounds** field for a memory section specifies `Each variable`, the code generator will surround each definition in a contiguous block of definitions with the comment, prepragma, and postpragma defined for the section.

If the **Pragma surrounds** field for a memory section specifies **All variables**, the code generator will insert the comment and prepragma for the section before the first definition in a contiguous block of custom storage class data definitions, and the postpragma after the last definition in the block.

Note: Specifying **All variables** affects only custom storage class data definitions. For other definition categories, the code generator surrounds each definition separately regardless of the value of **Pragma surrounds**.

Include Identifier Names in Pragmas

When pragmas surround each separate definition in a contiguous block, you can include the character vector `%<identifier>` in a pragma. The character vector must appear without surrounding quotes.

- When `%<identifier>` appears in a prepragma, the code generator will substitute the identifier from the subsequent function or data definition.
- When `%<identifier>` appears in a postpragma, the code generator will substitute the identifier from the previous function or data definition.

You can use `%<identifier>` with pragmas *only* when pragmas to surround each variable. The `Validate` phase will report an error if you violate this rule.

Note: Although `%<identifier>` looks like a TLC variable, it is not: it is just a keyword that directs the code generator to substitute the applicable data definition identifier when it outputs a pragma. TLC variables cannot appear in pragma specifications in the **Memory Section** pane.

Specify 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 other definition categories.

Apply Memory Sections

Assign Memory Sections to Custom Storage Classes

To assign a memory section to a custom storage class,

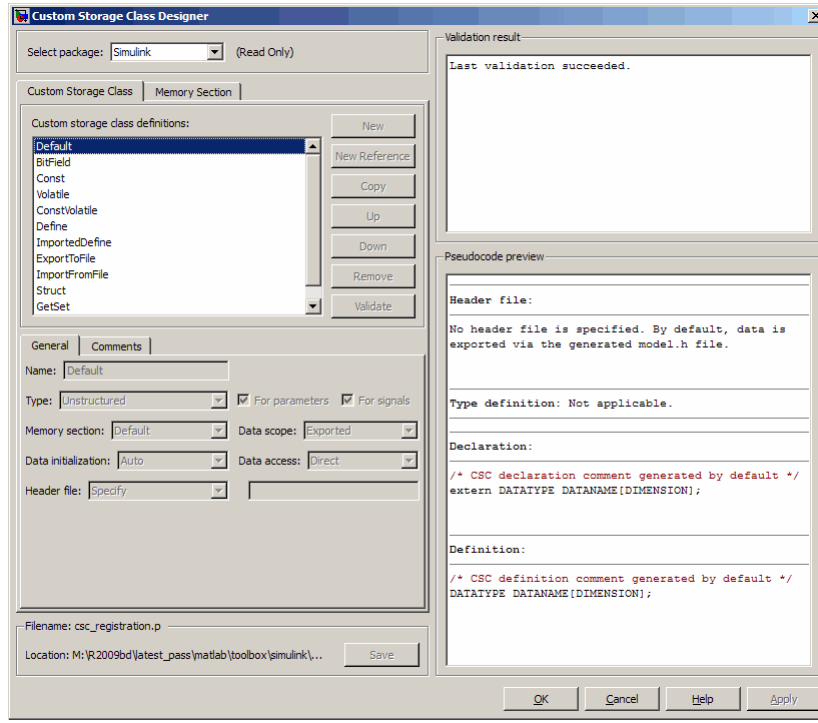
1. Choose **View > Model Explorer** in the model window.

   The Model Explorer appears.

2. Choose **Tools > Custom Storage Class Designer** in the Model Explorer window.

   A notification box appears that states **Please Wait ... Finding Packages**. After a brief pause, the notification box closes and the Custom Storage Class Designer appears.

3. Select the **Custom Storage Class** tab. The **Custom Storage Class** pane initially looks like this:



4. Use the **Select package** field to select a writable package. The rest of this section assumes that you have selected a writable package.
5. Select the desired custom storage class in the **Custom storage class definitions** pane.
6. Select the desired memory section from the **Memory section** pull-down.
7. Click **Apply** to apply changes to the open copy of the model; **Save** to apply changes and save them to disk; or **OK** to apply changes, save changes, and close the Custom Storage Class Designer.

Generated code for data definitions in the specified custom storage class are enclosed in the pragmas of the specified memory section. The pragmas can surround contiguous blocks of definitions or each definition separately, as described in “Surround Individual Definitions with Pragmas” on page 27-5. For more information, see “Design Custom Storage Classes and Memory Sections” on page 23-34.

Note: The code generator does not generate a `pragma` around definitions or declarations for data that has the following built-in storage classes:

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

The code generator treats data with these built-in storage classes like custom storage classes without a specified memory section.

Apply Memory Sections to Model-Level Functions and Internal Data

The table shows the categories of model-level functions to which you can apply memory sections.

Function Category	Functions Included in Memory Section
Initialize/Terminate functions	Initialize/Start
	Terminate
Execution functions	Step functions
	Run-time initialization
	Derivative
	Enable
	Disable
Shared utility functions	• Shared utility functions, such as those generated for model references. • Subfunctions, such as those generated by Simulink Coder for intrinsic math utilities, Stateflow graphical functions, and MATLAB subfunctions.

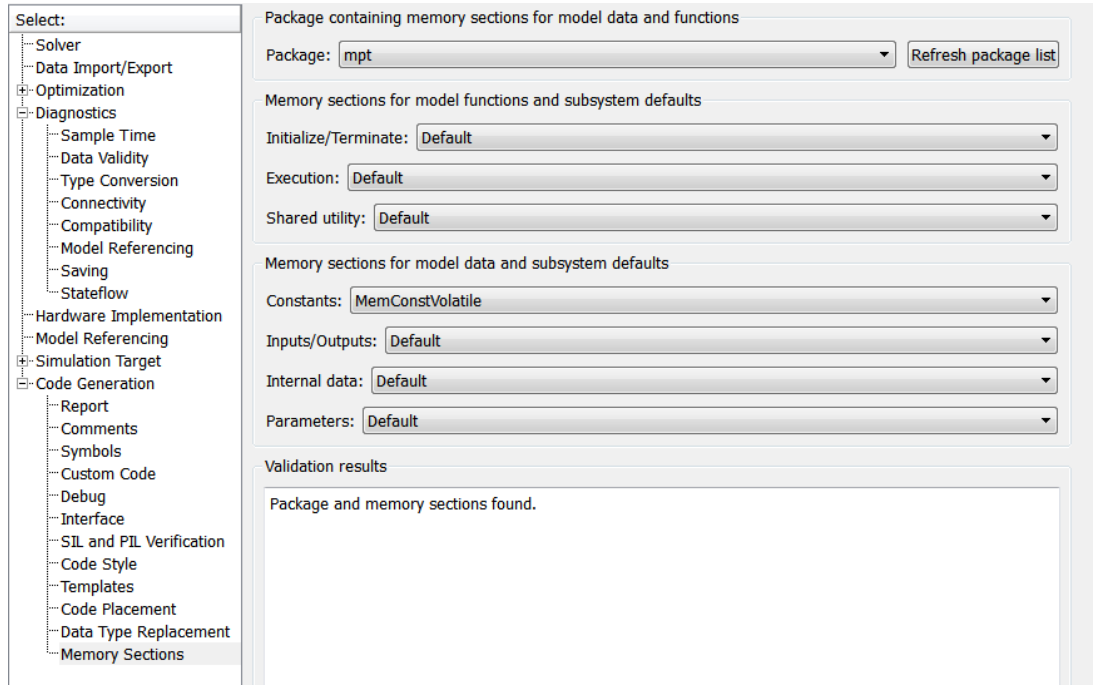
The table shows the categories of internal data to which you can apply memory sections. The specified memory section applies to the corresponding global data structures in the generated code. For basic information about the global data structures, see "Default Data Structures in the Generated Code" (Simulink Coder).

Data Category	Data Included in Memory Section
Constants	Constant block parameters
	Block inputs and outputs that have constant values
Input/Output	Root inputs
	Root outputs
Internal data	Block input and output signals
	DWork vectors
	Zero-crossings
Parameters	Block parameters

Memory section specifications for model-level functions and internal data apply to the top level of the model and to its subsystems. However, these specifications are not applicable to atomic subsystems that contain overriding memory section specifications, as described in “Apply Memory Sections to Atomic Subsystems” on page 27-10.

To specify memory sections for model-level functions or internal data,

1. Open the Configuration Parameters dialog box and select **Code Generation** > **General**.
2. Specify the **System target file** as an ERT target, such as `ert.tlc` .
3. Select **Memory Sections**. The **Memory Sections** pane looks like this:



4 Initially, the **Package** field specifies `---None---` and the pull-down lists only built-in packages. If you have defined packages of your own, click **Refresh package list**. This action adds 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 that you want to apply to that category. Accepting or specifying `Default` omits specifying memory section for that category.

7 Click **Apply** to save changes to the package and memory section selections.

Apply Memory Sections to Atomic Subsystems

For atomic subsystem whose **Function packaging** is **Nonreusable function** or **Reusable function**, you can specify memory sections for functions and internal data that exist in that code interface packaging. Such specifications override model-level memory section specifications. Such overrides apply only to the atomic subsystem itself,

not to subsystems within it. Subsystems of an atomic subsystem inherit memory section specifications from the containing model, *not* from the containing atomic subsystem.

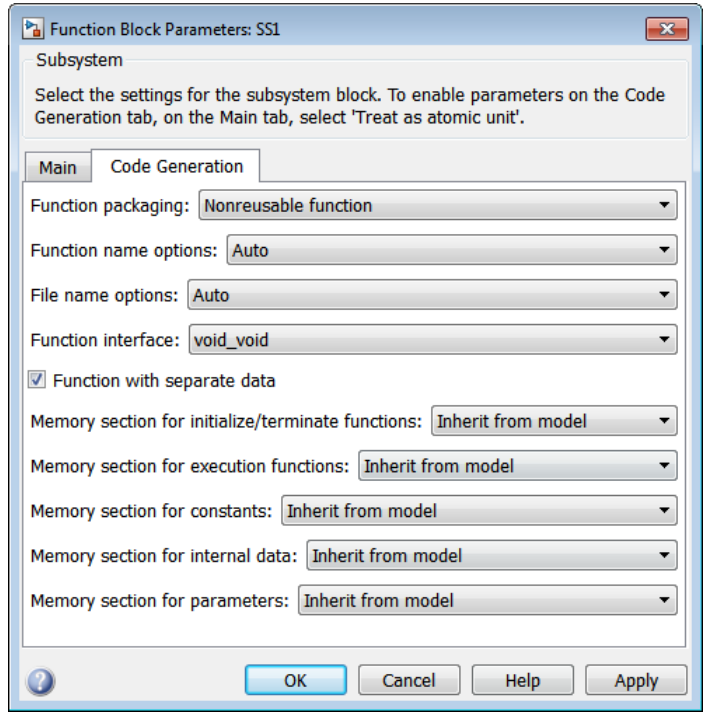
The memory section that you specify for internal data applies to the corresponding global data structures in the generated code. For basic information about the global data structures generated for atomic subsystems, see "Default Data Structures in the Generated Code" (Simulink Coder).

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. Select the **Treat as atomic unit** checkbox. If it is not selected, you cannot specify memory sections for the subsystem.

   For an atomic system, on the **Code Generation** tab, you can use the **Function packaging** field to control the format of the generated code.
4. Specify **Function packaging** as `Nonreusable function` or `Reusable function`. Otherwise, you cannot specify memory sections for the subsystem.
5. If the **Function packaging** is `Reusable function` and you want separate data, check **Function with separate data**.

   The **Code Generation** tab now shows applicable memory section options. The available options depend on the values of **Function packaging** and the **Function with separate data** check box. When the former is `Nonreusable function` and the latter is checked, the pane looks like this:



6 In the pull-down for each available definition category, specify the memory section that you want to apply to that category.

- Selecting `Inherit from model` inherits the corresponding selection from the model level (not parent subsystem).
- Selecting `Default` specifies that the category does not have an associated memory section, overriding model-level specifications for that category.

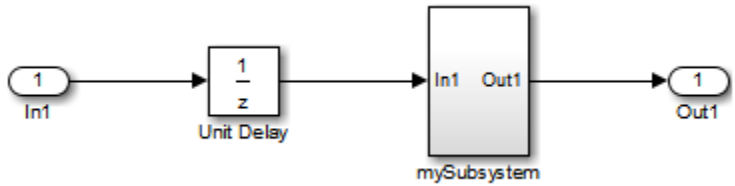
7 Click **Apply** to save changes, or **OK** to save changes and close the dialog box.

Caution: If you use **Build This Subsystem** or **Build Selected 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 building subsystems, see "Generate Code and Executables for Individual Subsystem" (Simulink Coder).

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.

Generated Code with Memory Sections

The next figures show an ERT-based Simulink model that defines one subsystem, `mySubsystem`, and then the contents of that subsystem.



Assume that the subsystem is atomic. On the **Code Generation** tab, the **Function packaging** parameter is `Reusable function`. Memory sections have been created and assigned as shown in the next two tables; here, data memory sections specify **Pragma surrounds** to be `Each variable`.

Model-Level Memory Section Assignments and Definitions

Section Assignment	Section Name	Field Name	Field Value
Input/Output	`MemSect1`	Prepragma	`#pragma IO-begin`
		Postpragma	`#pragma IO-end`
Internal data	`MemSect2`	Prepragma	`#pragma InData-begin(%<identifier>)`
		Postpragma	`#pragma InData-end`
Parameters	`MemSect3`	Prepragma	`#pragma Parameters-begin`
		Postpragma	`#pragma Parameters-end`

Section Assignment	Section Name	Field Name	Field Value
Initialize/ Terminate	MemSect4	Prepragma	#pragma InitTerminate-begin
		Postpragma	#pragma InitTerminate-end
Execution functions	MemSect5	Prepragma	#pragma ExecFunc-begin(%<identifier>)
		Postpragma	#pragma ExecFunc-end(%<identifier>)

Subsystem-Level Memory Section Assignments and Definitions

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

Given the preceding specifications and definitions, the code generator would create the following code, with minor variations depending on the current version of the Target Language Compiler.

Model-Level Data Structures

```
#pragma IO-begin
ExternalInputs_mySample mySample_U;
#pragma IO-end

#pragma IO-begin
ExternalOutputs_mySample mySample_Y;
#pragma IO-end

#pragma InData-begin(mySample_B)
BlockIO_mySample mySample_B;
#pragma InData-end

#pragma InData-begin(mySample_DWork)
D_Work_mySample mySample_DWork;
#pragma InData-end

#pragma InData-begin(mySample_M_)
RT_MODEL_mySample mySample_M_;
#pragma InData-end

#pragma Parameters-begin
```

```
Parameters_mySample mySample_P = {
  1.0 , {2.3}
};
#pragma Parameters-end
```

Model-Level Functions

```
#pragma ExecFunc-begin(mySample_step)
void mySample_step(void)
{
 mySample_B.UnitDelay = mySample_DWork.UnitDelay_DSTATE;
 mySample_mySubsystem();
 mySample_DWork.UnitDelay_DSTATE = mySample_U.In1;
}
#pragma ExecFunc-end(mySample_step)

#pragma InitTerminate-begin
void mySample_initialize(void)
{
  mySample_U.In1 = 0.0;
  mySample_Y.Out1 = 0.0;
  mySample_DWork.UnitDelay_DSTATE = mySample_P.UnitDelay_InitialCondition;
}
#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:

```
#pragma DATA_SEC(mySample_mySubsystem, “FAST_RAM”)
void mySample_mySubsystem(void)
{
 mySample_Y.Out1 = mySample_P.Gain_Gain * mySample_B.UnitDelay;
}
```

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:

```
#pragma ExecFunc-begin(mySample_mySubsystem)
void mySample_mySubsystem(void)
{
 mySample_Y.Out1 = mySample_P.Gain_Gain * mySample_B.UnitDelay;
}
```

```
#pragma ExecFunc-end(mySubsystem)
```

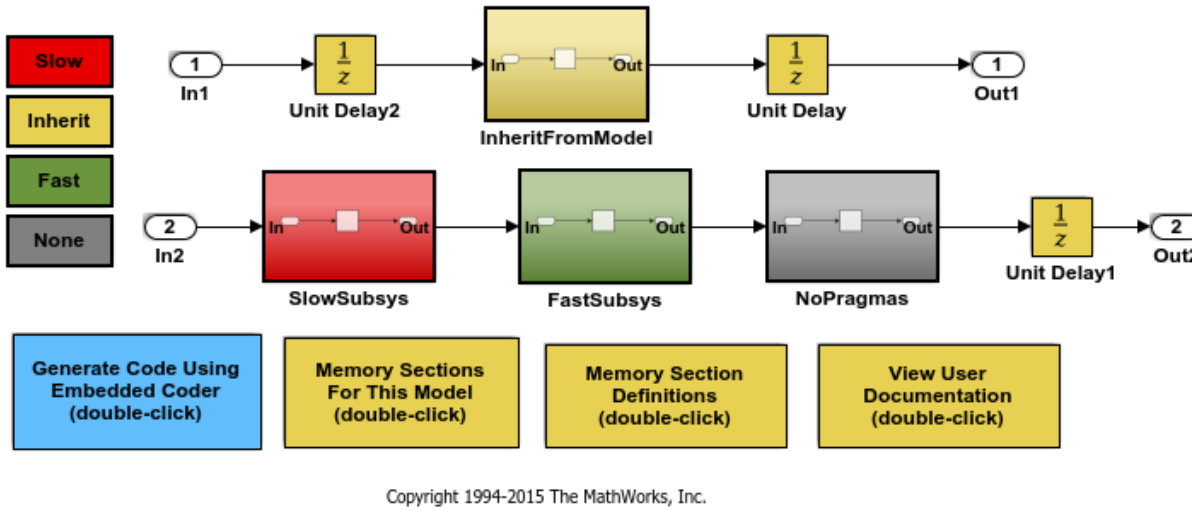
Insert Pragmas for Functions and Data in Generated Code

This model shows how to insert pragmas for functions and data in generated code.

Explore Example Model

Open the example model.

```
open_system('rtwdemo_memsec')
```



Instructions

1. Learn about memory sections by clicking the documentation link in the model.
2. View the memory sections in the ECoderDemos package by clicking the button in the model and then selecting the **Memory Sections** tab.
3. View the memory sections selected for this model by clicking the button in the model. The model-level settings are also the default settings for atomic subsystems.
4. Open the SubSystem Parameters dialog for the subsystems to see the memory section settings for each of the atomic subsystems in the model.

5 Generate code by clicking the button in the model. An HTML report is displayed automatically. Inspect the data and function definitions in the `.c` files and observe how the generated pragmas correspond to the specified memory sections.

Documenting Use of Pragmas with Simulink Report Generator

If you need to report on the use of pragmas to define memory sections, for example to satisfy the modeling guideline "hisl_0402: Use of custom #pragma to improve MISRA C:2012 compliance" (Simulink), you can customize the Simulink Report Generator report to include this information.

To add pragma information to your Simulink Report Generator report:

1 Open your Simulink Report Generator report. For more information on these reports, see "Getting Started with Simulink Report Generator" (Simulink Report Generator).

2 In the Library pane, under the **MATLAB** category, add the **Evaluate MATLAB Expression** component to your report.

3 Add code, similar to the following, to the **Expression to evaluate in the base workspace** text box. This code extracts pragma information from the model.

```
% Find data objects with PRAGMA information
% Note: only objects that are used should be returned

clear pragma;
pragma{1,1}  = {'Pragma Name'};
pragma{1,2}  = {'Pragma Type'};

% Data classes with PRAGMA Information
PragTypes = {'PragmaDemo.PragmaDef','OtherPragma.PragmaDef'};

allData = who;

for inx = 1 : length(allData)
   isPragma = strcmp(eval(['class(',allData{inx},')']),PragTypes);
   if ~isempty(find(isPragma))
      pragma{end+1,1} = allData{inx};
      pragma{end,2}   = PragTypes{isPragma};
   end
end
```

4 Select the **Evaluate this expression if there is an error** check box.

5 In the field under the check box, add code similar to the following:

```
% The exception generated above can be referenced by evalException
% See the default code below for an example
warningMessageLevel = 2;
displayWarningMessage = true;
failGenerationWithException = false;
failGenerationWithoutException = false;

if (displayWarningMessage)
    rptgen.displayMessage(sprintf('%s : %s',...
    'Exception during eval', evalException.message),...
    warningMessageLevel);
end

if (failGenerationWithException)
    rptgen.displayMessage('Failed to evaluate', warningMessageLevel);
    rethrow(evalException);

elseif (failGenerationWithoutException)
    rptgen.displayMessage(sprintf('%s\n%s','Failed to evaluate', ...
    'Halting generation'), warningMessageLevel, false);
    rptgen.haltGenerate;
end
```

6 Click **File** > **Save** to save the report setup file.

For more information on adding a component that evaluates MATLAB commands, see "Add Report Content with Components" (Simulink Report Generator).

Related Examples

- "Control Data Representation by Applying Custom Storage Classes" on page 23-58
- "Design Custom Storage Classes and Memory Sections" on page 23-34
- "Declare Constant Data as Volatile Using Memory Sections" on page 27-19
- "Default Data Structures in the Generated Code" (Simulink Coder)

Declare Constant Data as Volatile Using Memory Sections

In the C language, the value of data declared with the storage type qualifier, `volatile`, can be read from memory and written back to memory when changed without compiler control or detection. Examples of use include variables for initialization at system power-up or for system clock updates.

You can add the `volatile` qualifier to type definitions generated in code for model constant block I/O, constant parameters, and ground data (zero representation). Note that if the input to a reusable subsystem is volatile data, the code generator casts away the `volatile` qualifier. The subsystem argument is the address of the volatile data.

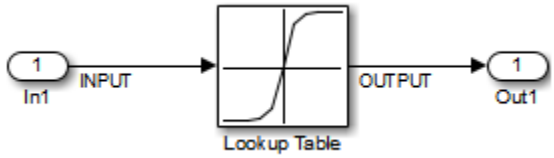
To add the `volatile` qualifier to type definitions, you must configure your model as follows:

- Specify an ERT target
- Set the memory section for constant data to `MemVolatile` or `MemConstVolatile`

If you choose to add the `volatile` qualifier to type definitions in your generated code, note the following:

- If constant data that is qualified with `volatile` is passed by pointer, the code generator casts away the volatility. This occurs because generated functions assume that data values do not change during execution and, therefore, pass their arguments as `const *` (not `const volatile *`).
- If a variable must be declared `const` and you specify `MemVolatile`, the code generator declares the variable with the `const` and `volatile` qualifiers.
- If you set **Constants** to `MemConst` or `MemConstVolatile`, and a variable cannot be declared as constant data, a TLC warning appears and the code generator does not qualify the variable with `const`.

Consider the following simple lookup table model.



1. On the Configuration Parameters dialog box, In the **Code Generation** pane, set **System target file** to `ert.tlc`.
2. In the **Code Generation** > **Memory Sections** pane, set **Package** to `Simulink`, and **Constants** to `MemConstVolatile`.
3. Open the Signal Properties dialog box for signal `INPUT`. On the **Code Generation** tab, set **Storage class** to `ExportedGlobal` for storing state in a global variable.
4. Generate code. You should see the `volatile` qualifier in the generated files *`model`*`_data.c` and *`model`*`.h`.

   `model``_data.c`

```
/* Constant parameters (auto storage) */
/* ConstVolatile memory section */
const volatile ConstParam_simple_lookup simple_lookup_ConstP = {
  /* Expression: [-5:5]
   * Referenced by: '<Root>/Lookup Table'
   */
  { -5.0, -4.0, -3.0, -2.0, -1.0, 0.0, 1.0, 2.0, 3.0, 4.0, 5.0 },

  /* Expression: tanh([-5:5])
   * Referenced by: '<Root>/Lookup Table'
   */
  { -0.99990920426259511, -0.999329299739067,
    -0.99505475368673046, -0.9640275800758169,
    -0.76159415595576485, 0.0, 0.76159415595576485,
    0.9640275800758169, 0.99505475368673046,
    0.999329299739067, 0.99990920426259511 }
};
```

   `model``.h`

```
/* Real-time Model Data Structure */
struct RT_MODEL_simple_lookup {
  const char_T * volatile errorStatus;
};

/* Constant parameters (auto storage) */
extern const volatile ConstParam_simple_lookup simple_lookup_ConstP;
```

   Also note in the *`model`*`.c` file that a typecast is inserted in the `rt_Lookup` function call, removing the `volatile` qualifier.

```
/* Lookup: '<Root>/Lookup Table' incorporates:
```

```
 *  Inport: '<Root>/In1'
 */
OUTPUT = rt_Lookup(((const real_T*)
  &simple_lookup_ConstP.LookupTable_XData[0]), 11, INPUT, ((
  const real_T*) &simple_lookup_ConstP.LookupTable_YData[0]));
```

Related Examples

- “Control Data Representation by Applying Custom Storage Classes” on page 23-58
- “Default Data Structures in the Generated Code” (Simulink Coder)

Code Generation

Configuration for Simulink Coder

Code Generation Configuration

When you are ready to generate code for a model, you can modify the model configuration parameters specific to code generation. The code generation parameters determine how the code generator produces code and builds an executable program from your model.

The model configuration parameters for code generation are in the **Code Generation** and **Optimization** panes in the Configuration Parameters dialog box. The content of the **Code Generation** pane and its subpanes can change depending on the target that you specify. Some configuration options are available only with the Embedded Coder product. The **Optimization** pane includes code generation parameters that help to improve the performance of the generated code.

Your application objectives can include a combination of these code generation objectives: debugging, traceability, execution efficiency, and safety precaution. There are tradeoffs associated with these configuration choices, such as execution speed and memory usage. To help configure a model to achieve your application objectives, use the Model Advisor and the Code Generation Advisor.

Open the Model Configuration for Code Generation

To modify the model configuration parameters for code generation, open the **Code Generation** pane. There are several different ways to open the **Code Generation** pane from the Simulink editor:

- To open the Configuration Parameters dialog box, click the model configuration parameters icon.

  Then click **Code Generation** in the **Select** (left) pane.
- From the **Simulation** menu, select **Model Configuration Parameters**. When the Configuration Parameters dialog box opens, click **Code Generation** in the **Select** (left) pane.
- From the **Code** menu, select **C/C++ Code** > **Code Generation Options**.
- From the **View** menu in the model window, select **Model Explorer**, or from the MATLAB command line, type `daexplr` and press **Enter**. In the Model Explorer,

expand the node for the current model in the left pane and click `Configuration (active)`. Click elements in the middle pane to display the corresponding parameters in the right pane.

Note In a Configuration Parameters dialog box, when you change the value of a check box, menu selection, or edit field, the white background of the element changes color to indicate that you made an unsaved change. When you click **OK**, **Cancel**, or **Apply**, the background resets to white.

Configuration Tools

To help you configure your model for code generation and to check your configuration against your code generation objectives, Simulink Coder and Embedded Coder provide several tools.

Goal	Approach	More Information
Automate configuration.	At the MATLAB command line, use the `set_param` function	"Configure Code Generation Parameters for Model Programmatically" on page 28-5
Configure your model for code generation quickly and easily (Embedded Coder).	Embedded Coder Quick Start tool	"Generate Code with the Quick Start Tool" on page 34-10
Use a template to create a model configured for code generation, ready for you to add your own blocks.	Code generation templates	• "Generate Code and Simulate Models in a Simulink Project" (Simulink Coder) • "Generate Code and Simulate Models in a Simulink Project"
To configure your model for code generation, use Simulink blocks and predefined or custom MATLAB scripts.	Code Generation Wizards blocks	"Configure and Optimize Model with Configuration Wizard Blocks" on page 29-21

Goal	Approach	More Information
Verify that your model meets standards and guidelines.	Model Advisor	"Select and Run Model Advisor Checks" (Simulink)
Verify that your model meets your application objectives.	Code Generation Advisor	"Application Objectives Using Code Generation Advisor" on page 28-9

Related Examples

- "Configuration Reuse" (Simulink)
- "Configure Code Generation Parameters for Model Programmatically" (Simulink Coder)
- "Application Objectives Using Code Generation Advisor" on page 28-9

Configure Code Generation Parameters for Model Programmatically

You can modify code generation parameters for the active configuration set in the Configuration Parameters dialog box or from the MATLAB command line. Use the command-line approach for creating a script that automates setting parameters for an established model configuration. The Configuration Parameters **All Parameters** tab provides parameter command-line names to use in scripts.

Modify Parameters to Support `Execution efficiency`

In this example, you modify the configuration parameters to support the Code Generation Advisor application objective, `Execution efficiency`.

Step 1. Open a model.

```
slexAircraftExample
```

Step 2. Get the active configuration set.

```
cs = getActiveConfigSet(model);
```

Step 3. Select the Generic Real-Time (GRT) target.

```
switchTarget(cs,'grt.tlc',[]);
```

Step 4. To optimize execution speed, modify parameters.

If your application objective is `Execution efficiency`, use `set_param` to modify these parameters:

```
set_param(cs,'MatFileLogging','off');
set_param(cs,'SupportNonFinite','off');
set_param(cs,'RTWCompilerOptimization','on');
set_param(cs,'OptimizeBlockIOStorage','on');
set_param(cs,'EnhancedBackFolding','on');
set_param(cs,'ConditionallyExecuteInputs','on')
set_param(cs,'DefaultParameterBehavior','Inlined');
set_param(cs,'BooleanDataType','on');
set_param(cs,'BlockReduction','on');
set_param(cs,'ExpressionFolding','on');
set_param(cs,'LocalBlockOutputs','on');
```

```
set_param(cs,'EfficientFloat2IntCast','on');
set_param(cs,'BufferReuse','on');
```

Step 5. Save the model configuration to a file.

Save the model configuration to a file, `'Exec_efficiency_cs.m'`, and view the parameter settings.

```
saveAs(cs,'Exec_Efficiency_cs');
dbtype Exec_Efficiency_cs 1:50
```

More About

- "Code Generation Configuration" on page 28-2
- "Application Objectives Using Code Generation Advisor" on page 28-9

Check Model and Configuration for Code Generation

You can use the Model Advisor checks to assess model readiness to generate code. To check and configure your model for code generation application objectives such as traceability or debugging, use the Code Generation Advisor.

For information about	See
Model Advisor	"Run Model Checks" (Simulink)
Code Generation Advisor	"Application Objectives Using Code Generation Advisor" (Simulink Coder)
Checks available with Simulink Coder	"Simulink Coder Checks" (Simulink Coder)
Checks available with Embedded Coder	"Embedded Coder Checks"

Check Mode for Code Efficiency with Model Advisor

To check model `rtwdemo_throttlecntrl` for code efficiency, use the Model Advisor.

1. Open `rtwdemo_throttlecntrl`. Save a copy as `throttlecntrl` in a writable location on your MATLAB path.
2. To start the Model Advisor, select **Analysis > Model Advisor > Model Advisor**. A dialog box opens showing the model system hierarchy.
3. Click `throttlecntrl` and then click **OK**. The Model Advisor window opens.
4. Expand **By Task > Code Generation Efficiency**. To check your model for code generation efficiency, use the checks in the folder . By default, checks that do not trigger an Update Diagram are selected. The checks available for code generation efficiency depend on whether you have a Simulink Coder or Embedded Coder license.
5. In the left pane, select the remaining checks, and then select **Code Generation Efficiency**.
6. In the right pane, select **Show report after run** and click **Run Selected Checks**. The report shows a **Run Summary** that flags check warnings.
7. Review the report. The warnings highlight issues that impact code efficiency. For more information about the report, see "View Model Advisor Reports" (Simulink).

Check Model During Code Generation with Code Generation Advisor

To review a model as part of the code generation process, use the Code Generation Advisor.

1. To specify your code generation objectives, on the **Configuration Parameters > Code Generation** pane, choose a value for the **Select objective** parameter.
2. On the **Configuration Parameters > Code Generation > General** pane, select one of the following from **Check model before generating code**:
   - `On (proceed with warnings)`
   - `On (stop for warnings)`
3. If you want to only generate code, select **Generate code only**. Otherwise clear the check box to build an executable.
4. Apply your changes, and then click **Generate Code/Build**. The Code Generation Advisor starts and reviews the top model and subsystems.

   If the Code Generation Advisor issues failures or warnings, and you specified:

   - `On (proceed with warnings)` — The Code Generation Advisor window opens while the build process proceeds. After the build process is complete, you can review the results.
   - `On (stop for warnings)` — The build process halts and displays the Diagnostic Viewer. To continue, you must review and resolve the Code Generation Advisor results or clear the **Check model before generating code** parameter.
5. In the Code Generation Advisor window, review the results by selecting a check from the left pane. The results for that check display in the right pane.
6. After reviewing the check results, you can choose to fix warnings and failures as described in "Fix a Model Check Warning or Failure" (Simulink).

Note: When you specify an efficiency or Safety precaution objective, the Code Generation Advisor includes additional checks. When you make changes to one of these additional checks, previous check results can potentially become invalid and need to be rerun.

For more information, see "Set Objectives — Code Generation Advisor Dialog Box" (Simulink Coder)

Related Examples

- "Select and Run Model Advisor Checks" (Simulink)
- "Application Objectives Using Code Generation Advisor" on page 28-9

Application Objectives Using Code Generation Advisor

In this section...

Consider how your application objectives, such as efficiency, traceability, and safety, map to code generation options in a model configuration set. Parameters that you set in the **Solver**, **Data Import/Export**, **Diagnostics**, and **Code Generation** panes in the Configuration Parameters dialog box specify the behavior of a model in simulation and the code generated for the model.

Before generating code, or as part of the code generation process, you can use the Code Generation Advisor to review a model . When you choose to review a model before generating code, you specify which model, subsystem, or referenced model the Code Generation Advisor reviews. When you choose to review a model as part of the code generation process, the Code Generation Advisor reviews the entire system. The Code Generation Advisor uses the information presented in "Recommended Settings Summary for Model Configuration Parameters" to determine the parameter values that meet your objectives. When there is a conflict between multiple objectives, the higher-priority objective takes precedence.

Setting code generation objectives, and then running the Code Generation Advisor provides information on how to meet code generation objectives for your model. The Code Generation Advisor does not alter the generated code. You can use the Code Generation Advisor to make the suggested changes to your model. The generated code is changed only after you modify your model and regenerate code. When you use the Code Generation Advisor to set code generation objectives and check your model, the generated code includes comments identifying which objectives you specified, the checks the Code Generation Advisor ran on the model, and the results of running the checks.

If a model uses a configuration reference (Simulink), you can run the Code Generation Advisor to review your configuration parameter settings. However, the Code Generation Advisor cannot modify the configuration parameter settings.

High-Level Code Generation Objectives

Depending on the type of application that your model represents, you are likely to have specific high-level code generation objectives. For example, safety and traceability might be more critical than efficient use of memory. If you have specific objectives, you can quickly configure your model to meet those objectives by selecting and prioritizing from these code generation objectives:

- Execution efficiency (all targets) — Configure code generation settings to achieve fast execution time.
- ROM efficiency (ERT-based targets) — Configure code generation settings to reduce ROM usage.
- RAM efficiency (ERT-based targets) — Configure code generation settings to reduce RAM usage.
- Traceability (ERT-based targets) — Configure code generation settings to provide mapping between model elements and code.
- Safety precaution (ERT-based targets) — Configure code generation settings to increase clarity, determinism, robustness, and verifiability of the code.
- Debugging (all targets) — Configure code generation settings to debug the code generation build process.
- MISRA C:2012 guidelines (ERT-based targets) — Configure code generation settings to increase compliance with MISRA C:2012 guidelines.
- Polyspace (ERT-based targets) — Configure code generation settings to prepare the code for Polyspace analysis.

If you select the MISRA C:2012 guidelines code generation objective, the Code Generation Advisor:

- Checks the model configuration settings for compliance with the MISRA C:2012 configuration setting recommendations.
- Checks for blocks that are not supported or recommended for MISRA C:2012 compliant code generation.

Configure Model for Code Generation Objectives Using Code Generation Advisor

This example shows how to use the Code Generation Advisor to check and configure your model to meet code generation objectives:

1. On the menu bar, select **Code** > **C/C++ Code** > **Code Generation Advisor**.
2. In the System Selector window, select the model or subsystem that you want to review, and then click **OK**.
3. In the Code Generation Advisor, on the **Code Generation Objectives** pane, select the code generation objectives from the drop-down list (GRT-based targets). As you select objectives, on the left pane, the Code Generation Advisor updates the list of checks it will run on your model. If your model is configured with an ERT-based target, more objectives are available.
4. Click **Run Selected Checks** to run the checks listed in the left pane of the Code Generation Advisor.
5. In the Code Generation Advisor window, review the results for **Check model configuration settings against code generation objectives** by selecting it from the left pane. The results for that check are displayed in the right pane.

   Check model configuration settings against code generation objectives triggers a warning for these issues:

   - Parameters are set to values other than the value recommended for the specified code generation objectives.
   - Selected code generation objectives differ from the objectives set in the model.

   Click **Modify Parameters** to set:

   - Parameters to the value recommended for the specified code generation objectives.
   - Code generation objectives in the model to the objectives specified in the Code Generation Advisor.
6. In the Code Generation Advisor window, review the results for the remaining checks by selecting them from the left pane. The results for the checks display in the right pane.
7. After reviewing the check results, you can choose to fix warnings and failures, as described in “Fix a Model Check Warning or Failure” (Simulink).

When you specify an efficiency or Safety precaution objective, the Code Generation Advisor includes additional checks. When you make changes to one of these additional checks, previous check results can potentially become invalid and need to be rerun.

Configure Model for Code Generation Objectives by Using Configuration Parameters Dialog Box

This example shows how to check and configure the code generation objectives in the Configuration Parameters dialog box:

1. Open the Configuration Parameters dialog box and select **Code Generation**.
2. Select or confirm selection of a System target file.
3. Specify the objectives using the **Select objectives** drop down list (GRT-based targets) or clicking **Set Objectives** button (ERT-based targets). Clicking **Set Objectives** opens the "Set Objectives — Code Generation Advisor Dialog Box" (Simulink Coder) dialog box.
4. Click **Check Model** to run the model checks. The Code Generation Advisor dialog box opens. The Code Generation Advisor uses the code generation objectives to determine which model checks to run.
5. On the left pane, the Code Generation Advisor lists the checks run on the model and the results. Click each warning to see the suggestions for changes that you can make to your model to pass the check.
6. Determine which changes to make to your model. On the right pane of the Code Generation Advisor, follow the instructions listed for each check to modify the model.

Simulink Coder Model Advisor Checks for Standards and Code Efficiency

To check that your model meets standards and is ready to generate code, you can use the Model Advisor checks available with Simulink Coder.

- To start the Model Advisor, in the model window, select **Analysis** > **Model Advisor** > **Model Advisor**.
- In the Model Advisor window, expand the **By Task** folder. The folder contains Model Advisor checks that you can run to help accomplish the task.

For more information about the Model Advisor, see "Run Model Checks" (Simulink).

The table summarizes the Simulink Coder Model Advisor checks that are available in the **By Task** folders.

By Task folder	Model Advisor checks
Code Generation Efficiency	"Identify blocks using one-based indexing" (Simulink Coder)
Modeling Standards for DO-178C/ DO-331	"Check solver for code generation" (Simulink Coder) "Check for blocks that have constraints on tunable parameters" (Simulink Coder) "Check sample times and tasking mode" (Simulink Coder)
Model Referencing	"Check for model reference configuration mismatch" (Simulink Coder) "Check for code generation identifier formats used for model reference" (Simulink Coder)

Related Examples

- "Select and Run Model Advisor Checks" (Simulink)
- "Embedded Coder Model Advisor Checks for Standards, Guidelines, and Code Efficiency" on page 29-12
- "Modeling Guidelines for Model Configuration" (Simulink Coder)

Configure Code Comments

Configure how the code generator inserts comments into generated code by modifying parameters on the **Code Generation > Comments** pane.

To...	Select...
Include comments in generated code	**Include comments** (Simulink Coder). Selecting this parameter allows you to select one or more auto generated comment types to be placed in the code.
Include comments that describe a block's code	**Simulink block / Stateflow object comments** (Simulink Coder). Selecting this parameter includes the comments before the block's code in the generating file.
Include MATLAB source code as comments	**MATLAB source code as comments** (Simulink Coder). Selecting this parameter inserts these comments preceding the associated generated code. The function signature is included in the function banner.
Include comments for eliminated blocks	**Show eliminated blocks** (Simulink Coder). Selecting this parameter includes comments for blocks that were eliminated as the result of optimizations, such as inlining parameters.
Include parameter comments regardless of the number of parameters	**Verbose comments for SimulinkGlobal storage class** (Simulink Coder). Selecting this parameter includes comments for parameter variable names and names of source blocks in the model parameter structure declaration in *`model`*`_prm.h`. If you do not select this parameter, parameter comments are generated if less than 1000 parameters are declared. This reduces the size of the generated file for models with a large number of parameters.

Note When you configure the code generator to produce code that includes comments, the code generator includes text for model parameters, block names, signal names, and Stateflow object names in the generated code comments. If the text includes characters that are unrepresented in the character set encoding for the model, the code generator replaces the characters with XML escape sequences. For example, the code generator replaces the Japanese full-width Katakana letter ア with the escape sequence `&#x30A2;`. For more information, see "Internationalization and Code Generation" (Simulink Coder).

Construction of Generated Identifiers

For GRT and RSim targets, the code generator automatically constructs identifiers for variables and functions in the generated code. These identifiers represent:

- Signals and parameters that have `Auto` storage class
- Subsystem function names that are not user defined
- Stateflow names

The components of a generated identifier include

- The root model name, followed by
- The name of the generating object (signal, parameter, state, and so on), followed by
- Unique *name-mangling* text

The code generator conditionally generates the name-mangling text to resolve potential conflicts with other generated identifiers.

To configure how the code generator names identifiers and objects, see:

- "Specify Identifier Length to Avoid Naming Collisions" on page 28-17
- "Specify Reserved Names for Generated Identifiers" on page 28-18

The code generator reserves certain words for its own use as keywords of the generated code language. For more information, see "Reserved Keywords" on page 28-19.

With an Embedded Coder license, you can specify parameters to control identifier formats, mangle length, scalar inlined parameters, and Simulink data object naming rules. For more information, see "Customize Generated Identifier Naming Rules" on page 36-15.

Identifier Name Collisions and Mangling

In identifier generation, a circumstance that would cause generation of two or more identical identifiers is called a *name collision*. When a potential name collision exists, unique *name-mangling* text is generated and inserted into each of the potentially conflicting identifiers. Each set of name-mangling characters is unique for each generated identifier.

Identifier Name Collisions with Referenced Models

Referenced models can introduce additional naming constraints. Within a model that uses referenced models, collisions between the names of the models cannot exist. When you generate code from a model that includes referenced models, the **Maximum identifier length** parameter must be large enough to accommodate the root model name and name-mangling text. A code generation error occurs if **Maximum identifier length** is too small.

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.

For more information on referenced models, see “Parameterize Instances of a Reusable Referenced Model” (Simulink).

Specify Identifier Length to Avoid Naming Collisions

The length of a generated identifier is limited by the **Maximum identifier length** parameter specified on the **Symbols** pane of the Configuration Parameters dialog box. The **Maximum identifier length** field allows you to limit the number of characters in function, type definition, and variable names. The default is 31 characters. This is also the minimum length you can specify. The maximum is 256 characters.

When there is a potential name collision between two identifiers, name-mangling text is generated. The text has the minimum number of characters required to avoid the collision. The other symbol components are then inserted. If **Maximum identifier length** is not large enough to accommodate full expansions of the other components, they are truncated. To avoid this outcome, it is good practice to:

- Avoid name collisions by not using default block names (for example, `Gain1`, `Gain2...`) when the model includes multiple blocks of the same type.
- For subsystems, make them atomic and reusable.
- Increase the **Maximum identifier length** parameter to accommodate the length of the identifier you expect to generate.

Specify Reserved Names for Generated Identifiers

You can specify a set of reserved keywords that the code generation process should not use, facilitating code integration where functions and variables from external environments are unknown in the Simulink model. To create a list of reserved names, open the Configuration Parameters dialog box. On the **Code Generation > Symbols** pane, enter the keywords in the “Reserved names” (Simulink Coder) field.

If your model contains MATLAB Function or Stateflow blocks, the code generation process can use the reserved names specified for those blocks if you select **Use the same reserved names as Simulation Target** (Simulink Coder) on the **Code Generation > Symbols** pane.

Reserved Keywords

In this section...

Generator keywords are reserved for internal use. Do not use them in models as identifiers or function names. Also avoid using C reserved keywords in models as identifiers or function names. If your model contains reserved keywords, code generation does not complete and an error message appears. To address the error, modify your model to use identifiers or names that are not reserved.

If you use the code generator to produce C++ code, your model must not contain the “Reserved Keywords for Code Generation” on page 28-20 nor the “C++ Reserved Keywords” on page 28-20.

Note: You can register additional reserved identifiers in the Simulink environment. For more information, see “Specify Reserved Names for Generated Identifiers” on page 28-18.

C Reserved Keywords

`auto`	`double`	`int`	`struct`
`break`	`else`	`long`	`switch`
`case`	`enum`	`register`	`typedef`
`char`	`extern`	`return`	`union`
`const`	`float`	`short`	`unsigned`
`continue`	`for`	`signed`	`void`
`default`	`goto`	`sizeof`	`volatile`
`do`	`if`	`static`	`while`

C++ Reserved Keywords

catch	friend	protected	try
class	inline	public	typeid
const_cast	mutable	reinterpret_cast	typename
delete	namespace	static_cast	using
dynamic_cast	new	template	virtual
explicit	operator	this	wchar_t
export	private	throw	

Reserved Keywords for Code Generation

abs	int8_T	MAX_uint8_T*	rtInf
asm	int16_T	MAX_uint16_T*	rtMinusInf
bool	int32_T	MAX_uint32_T*	rtNaN
boolean_T	int64_T	MAX_uint64_T	SeedFileBuffer
byte_T	INTEGER_CODE	MIN_int8_T*	SeedFileBufferLen
char_T	LINK_DATA_BUFFER_SIZE	MIN_int16_T*	single
cint8_T	LINK_DATA_STREAM	MIN_int32_T*	TID01EQ
cint16_T	localB	MIN_int64_T	time_T
cint32_T	localC	MODEL	true
creal_T	localDWork	MT	uint_T
creal32_T	localP	NCSTATES	uint8_T
creal64_T	localX	NULL	uint16_T
cuint8_T	localXdis	NUMST	uint32_T
cuint16_T	localXdot	pointer_T	uint64_T
cuint32_T	localZCE	PROFILING_ENABLED	UNUSED_PARAMETER
ERT	localZCSV	PROFILING_NUM_SAMPLES	USE_RTMODEL
false	matrix	real_T	VCAST_FLUSH_DATA
fortran	MAX_int8_T*	real32_T	vector

HAVESTDIO	MAX_int16_T*	real64_T	
id_t	MAX_int32_T*	RT	
int_T	MAX_int64_T	RT_MALLOC	
*Not reserved if you specify a replacement identifier.			

Code Generation Code Replacement Library Keywords

The list of code replacement library reserved keywords for your development environment varies depending on which libraries are registered. The list of available code replacement libraries varies depending on other installed products (for example, a target product), or if you used Embedded Coder to create and register custom code replacement libraries.

To generate a list of reserved keywords for libraries currently registered in your environment, use the following MATLAB function:

```
lib_ids = RTW.TargetRegistry.getInstance.getTflReservedIdentifiers()
```

This function returns an array of library keywords. Specifying the input argument is optional.

Note: To list the libraries currently registered in your environment, use the MATLAB command `crviewer`.

To generate a list of reserved keywords for a specific library that you are using to generate code, call the function passing the name of the library as displayed in the **Code replacement library** menu on the **Code Generation > Interface** pane of the Configuration Parameters dialog box. For example,

```
lib_ids = RTW.TargetRegistry.getInstance.getTflReservedIdentifiers('GNU C99 Extensions')
```

Here is a partial example of the function output:

```
>> lib_ids = RTW.TargetRegistry.getInstance.getTflReservedIdentifiers('GNU C99 Extensions')

lib_ids =

    'exp10'
    'exp10f'
    'acosf'
    'acoshf'
```

```
    'asinf'
    'asinhf'
    'atanf'
    'atanhf'
...
    'rt_lu_cplx'
    'rt_lu_cplx_sgl'
    'rt_lu_real'
    'rt_lu_real_sgl'
    'rt_mod_boolean'
    'rt_rem_boolean'
    'strcpy'
    'utAssert'
```

Note: Some of the returned keywords appear with the suffix `$N`, for example, `'rt_atan2$N'`. `$N` expands into the suffix `_snf` only if nonfinite numbers are supported. For example, `'rt_atan2$N'` represents `'rt_atan2_snf'` if nonfinite numbers are supported and `'rt_atan2'` if nonfinite numbers are not supported. As a precaution, you should treat both forms of the keyword as reserved.

Debug

In the Configuration Parameters dialog box, use parameters on the **Diagnostics** pane and debugging parameters on the **All Parameters** tab to configure a model such that the generated code and the build process are set for debugging. You can set parameters that apply to the model compilation phase, the target language code generation phase, or both.

Parameters in the following table will be helpful if you are writing TLC code for customizing targets, integrating legacy code, or developing new blocks.

To...	Select...
Display progress information during code generation in the MATLAB Command Window	"Verbose build" (Simulink Coder).Compiler output also displays.
Prevent the build process from deleting the *model*`.rtw` file from the build folder at the end of the build	"Retain .rtw file" (Simulink Coder). This parameter is useful if you are modifying the target files, in which case you need to look at the *model*`.rtw` file.
Instruct the TLC profiler to analyze the performance of TLC code executed during code generation and generate a report	"Profile TLC" (Simulink Coder). The report is in HTML format and can be read in your Web browser.
Start the TLC debugger during code generation	"Start TLC debugger when generating code" (Simulink Coder). Alternatively, enter the argument `-dc` for the "System target file" (Simulink Coder) parameter on the **Code Generation** pane. To start the debugger and run a debugger script, enter `-df filename` for **System target file**.
Generate a report containing statistics indicating how many times the code generator reads each	"Start TLC coverage when generating code" (Simulink Coder). Alternatively, enter the argument `-dg` for the **System Target File** parameter on the **Code Generation** pane.

To...	Select...
line of TLC code during code generation	

Halt a build if a user-supplied TLC file contains an `%assert` directive that evaluates to `FALSE`	"Enable TLC assertion" (Simulink Coder). Alternatively, you can use MATLAB commands to control TLC assertion handling.
	
	To set the flag on or off, use the `set_param` command. The default is off.
	
	`set_param(model, 'TLCAssertion', 'on|off')`
	
	To check the current setting, use `get_param`.
	
	`get_param(model, 'TLCAssertion')`

Detect loss of tunability	"Detect loss of tunability" (Simulink) on the **Diagnostics** > **Data Validity** pane. You can use this parameter to report loss of tunability when an expression is reduced to a numeric expression. This can occur if a tunable workspace variable is modified by Mask Initialization code, or is used in an arithmetic expression with unsupported operators or functions. Possible values are: • `none` — Loss of tunability can occur without notification. • `warning` — Loss of tunability generates a warning (default). • `error` — Loss of tunability generates an error. For a list of supported operators and functions, see "Tunable Expression Limitations" (Simulink Coder)

To...	Select...

Enable model verification (assertion) blocks	"Model Verification block enabling" (Simulink) on the **All Parameters**. Use this parameter to enable or disable model verification blocks such as Assert, Check Static Gap, and related range check blocks. The diagnostic applies to generated code as well as simulation behavior. For example, simulation and code generation ignore this parameter when model verification blocks are inside an S-function. Possible values are:
	• `User local settings`
	• `Enable All`
	• `Disable All`
	For Assertion blocks not disabled, generated code for a model includes one of the following statements, depending on the blocks input signal type (Boolean, real, or integer, respectively).
	`utAssert(input_signal);`
	`utAssert(input_signal != 0.0);`
	`utAssert(input_signal != 0);`
	By default, `utAssert` does not change generated code. For assertions to abort execution, you must enable them by specifying the following `make_rtw` command for **Code Generation >** "Make command" (Simulink Coder) parameter:
	`make_rtw OPTS="-DDOASSERTS"`
	Use the following variant if you want triggered assertions to print the assertion statement instead of aborting execution:
	`make_rtw OPTS="-DDOASSERTS -DPRINT_ASSERTS"`
	utAssert is defined as `#define utAssert(exp) assert(exp)`.
	To customize assertion behavior, provide your own definition of `utAssert` in a handwritten header file that overrides the default `utAssert.h`. For details on how to include a customized header file in generated code, see "Integrate

To...	Select...
	External Code by Using Model Configuration Parameters" (Simulink Coder). When running a model in accelerator mode, the Simulink engine calls back to itself to execute assertion blocks instead of using generated code. Thus, user-defined callbacks are still called when assertions fail.

For more information about the TLC debugging options, see Debugging on "Target Language Compiler" (Simulink Coder). Also, consider using the Model Advisor as a tool for troubleshooting model builds.

More About

- "Tunable Expression Limitations" (Simulink Coder)
- "Integrate External Code by Using Model Configuration Parameters" (Simulink Coder)
- "Target Language Compiler" (Simulink Coder)

Configuration in Embedded Coder

Configure Model for Code Generation Objectives by Using Code Generation Advisor

In this section...

Consider how your application objectives, such as efficiency, traceability, and safety, map to code generation parameters in a model configuration set. Parameters that you set in the **Solver**, **Data Import/Export**, **Diagnostics**, and **Code Generation** panes in the Configuration Parameters dialog box specify the behavior of a model in simulation and the code generated for the model.

Before generating code, or as part of the code generation process, you can use the Code Generation Advisor to review a model. When you choose to review a model before generating code, you specify which model, subsystem, or referenced model the Code Generation Advisor reviews. When you choose to review a model as part of the code generation process, the Code Generation Advisor reviews the entire system. The Code Generation Advisor uses the information presented in "Recommended Settings Summary for Model Configuration Parameters" to determine the parameter values that meet your objectives. When there is a conflict between multiple objectives, the higher-priority objective takes precedence.

Setting code generation objectives, and then running the Code Generation Advisor provides information on how to meet code generation objectives for your model. The Code Generation Advisor does not alter the generated code. You can use the Code Generation Advisor to make the suggested changes to your model. The generated code is changed only after you modify your model and regenerate code. When you use the Code Generation Advisor to set code generation objectives and check your model, the generated code includes comments identifying which objectives you specified, the checks that the Code Generation Advisor ran on the model, and the results of running the checks.

If a model uses a configuration reference (Simulink), you can run the Code Generation Advisor to review your configuration parameter settings. but the Code Generation Advisor cannot modify the configuration parameter settings.

High-Level Code Generation Objectives

Depending on the type of application that your model represents, you are likely to have specific high-level code generation objectives. For example, safety and traceability are more critical than efficient use of memory. If you have specific objectives, you can quickly configure your model to meet those objectives by selecting and prioritizing from these code generation objectives:

- Execution efficiency (all targets) — Configure code generation settings to achieve fast execution time.
- ROM efficiency (ERT-based targets) — Configure code generation settings to reduce ROM usage.
- RAM efficiency (ERT-based targets) — Configure code generation settings to reduce RAM usage.
- Traceability (ERT-based targets) — Configure code generation settings to provide mapping between model elements and code.
- Safety precaution (ERT-based targets) — Configure code generation settings to increase clarity, determinism, robustness, and verifiability of the code.
- Debugging (all targets) — Configure code generation settings to debug the code generation build process.
- MISRA C:2012 guidelines (ERT-based targets) — Configure code generation settings to increase compliance with MISRA C:2012 guidelines.
- Polyspace (ERT-based targets) — Configure code generation settings to prepare the code for Polyspace analysis.

If you select the MISRA C:2012 guidelines code generation objective, the Code Generation Advisor:

- Checks the model configuration settings for compliance with the MISRA C:2012 configuration setting recommendations.
- Checks for blocks that are not supported or recommended for MISRA C:2012 compliant code generation.

Specify Objectives in Referenced Models

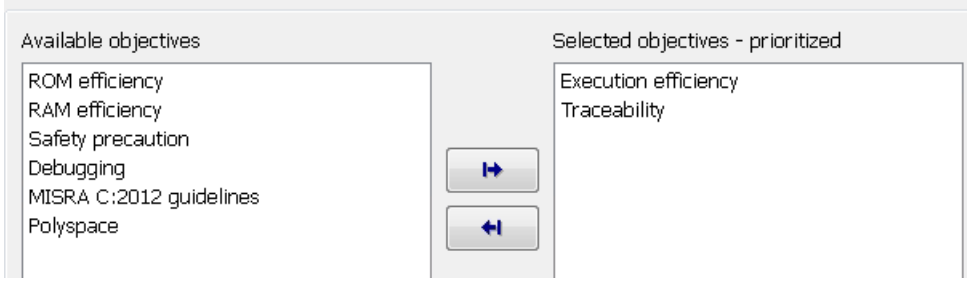
When you check a model during the code generation process, you must specify the same objectives in the top model and referenced models. If you specify different objectives for the top model and referenced model, the build process generates an error.

To specify different objectives for the top model and each referenced model, review the models separately without generating code.

Configure Model Using Code Generation Advisor

This example shows how to use the Code Generation Advisor to check and configure your model to meet code generation objectives:

1. On the menu bar, select **Code > C/C++ Code > Code Generation Advisor**.
2. In the System Selector window, select the model or subsystem that you want to review, and then click **OK**.
3. In the Code Generation Advisor, on the **Code Generation Objectives** pane, select the code generation objectives. As you select objectives, on the left pane, the Code Generation Advisor updates the list of checks it runs on your model. If your model is configured with an ERT-based target, more objectives are available. For this example, the model is configured with an ERT-based target. If your objectives are execution efficiency and traceability, in that priority, do the following:
   a. In **Available objectives**, double-click `Execution efficiency`. `Execution efficiency` is added to **Selected objectives - prioritized**.
   b. In **Available objectives**, double-click `Traceability`. `Traceability` is added to **Selected objectives - prioritized** under `Execution efficiency`.



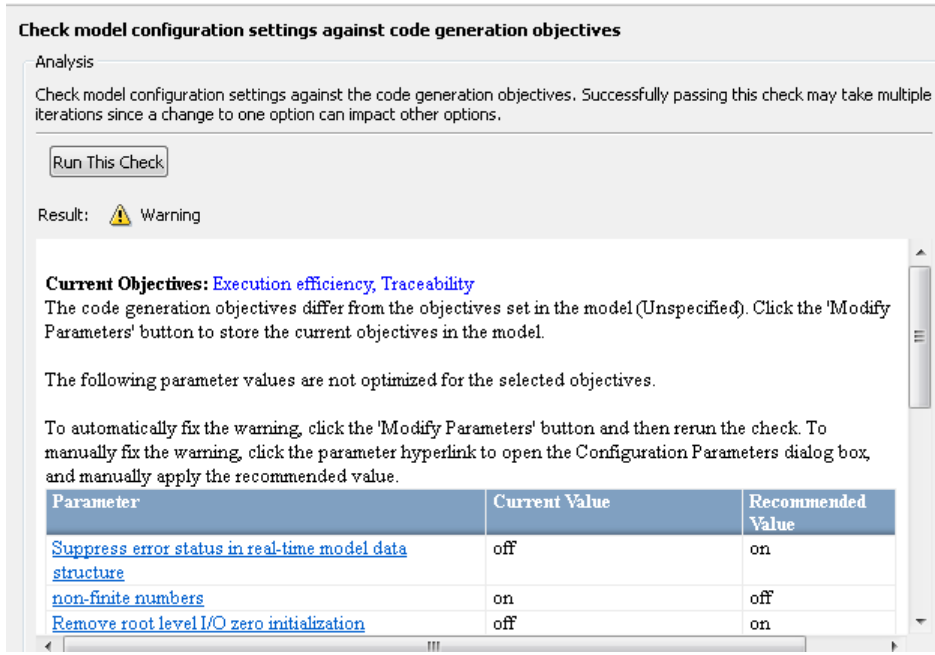
4. To run the checks listed in the left pane of the Code Generation Advisor, click **Run Selected Checks**.
5. In the Code Generation Advisor window, review the results for **Check model configuration settings against code generation objectives** by selecting it from the left pane. The results for that check are displayed in the right pane.

Check model configuration settings against code generation objectives triggers a warning for these issues:

- Parameters are set to values other than the value recommended for the specified code generation objectives.
- Selected code generation objectives differ from the objectives set in the model.

Click **Modify Parameters** to set:

- Parameters to the value recommended for the specified code generation objectives.
- Code generation objectives in the model to the objectives specified in the Code Generation Advisor.



6 In the Code Generation Advisor window, review the results for the remaining checks by selecting them from the left pane. The results for the checks display in the right pane.

7 After reviewing the check results, you can choose to fix warnings and failures, as described in "Fix a Model Check Warning or Failure" (Simulink).

When you specify an efficiency or Safety precaution objective, the Code Generation Advisor includes additional checks. When you make changes to one of these additional checks, previous check results can potentially become invalid and need to be rerun.

Configure Model for Code Generation Objectives by Using Configuration Parameters Dialog Box

This example shows how to configure and check your model to meet code generation objectives via the Configuration Parameters dialog box:

1 Open the Configuration Parameters dialog box. Select **Code Generation**.

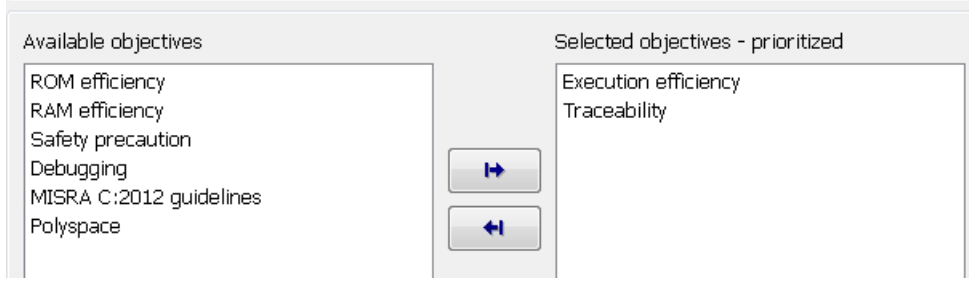
2 Specify a system target file. If you specify an ERT-based target, more objectives are available. For this example, choose an ERT-based target such as `ert.tlc`.

3 Click **Set Objectives**.

4 In the "Set Objectives — Code Generation Advisor Dialog Box" (Simulink Coder), specify your objectives. For example, if your objectives are execution efficiency and traceability, in that priority, do the following:

   a In **Available objectives**, double-click `Execution efficiency`. `Execution efficiency` is added to **Selected objectives - prioritized**.

   b In **Available objectives**, double-click `Traceability`. `Traceability` is added to **Selected objectives - prioritized** under `Execution efficiency`.

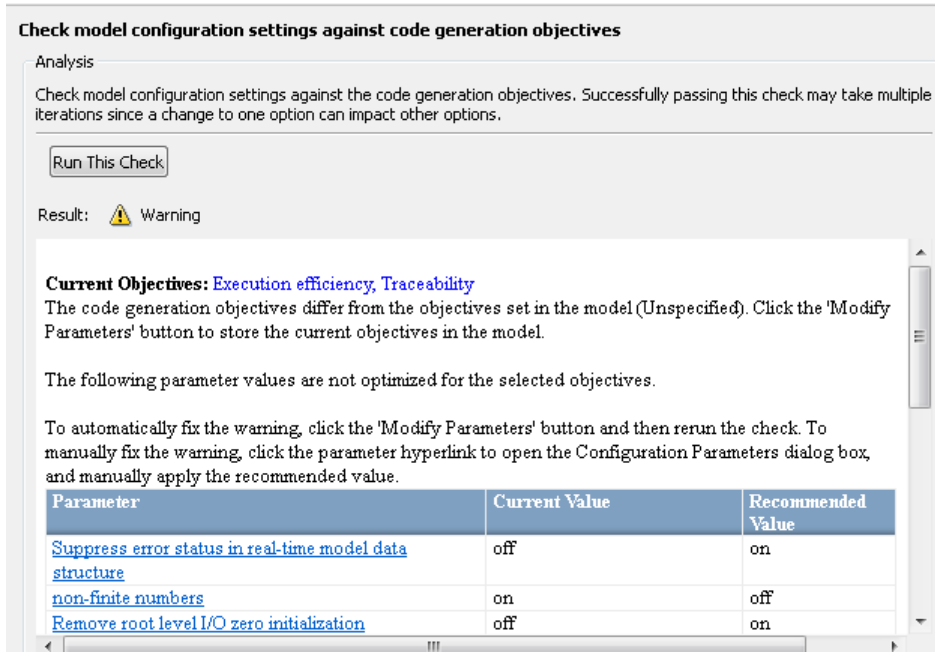


   c To accept the objectives, click **OK**. In the Configuration Parameters dialog box, **Code Generation** > **General** > **Prioritized objectives** is updated.

5 On the **Configuration Parameters** > **Code Generation** > **General** pane, click **Check Model**.

6 In the System Selector window, select the model or subsystem that you want to review, and then click **OK**. The Code Generation Advisor opens and reviews the model or subsystem that you specified.

7 In the Code Generation Advisor window, review the results by selecting a check from the left pane. The results for that check display in the right pane.



8 After reviewing the check results, you can choose to fix warnings and failures, as described in "Fix a Model Check Warning or Failure" (Simulink).

When you specify an efficiency or Safety precaution objective, the Code Generation Advisor includes additional checks. When you make changes to one of these additional checks, previous check results can potentially become invalid and need to be rerun.

For more information, see "Set Objectives — Code Generation Advisor Dialog Box" (Simulink Coder)

Related Examples

- "Configure Code Generation Objectives Programmatically" on page 29-9

- “Recommended Settings Summary for Model Configuration Parameters”
- “Code Generation Advisor Checks” (Simulink Coder)

Configure Code Generation Objectives Programmatically

This example shows how to configure code generation objectives by writing a MATLAB script or entering commands at the command line.

1. Specify a system target file. If you specify an ERT-based target, more objectives are available. For this example, specify `ert.tlc`. *`model_name`* is the name or handle to the model.

   ```
   set_param(model_name, 'SystemTargetFile', 'ert.tlc');
   ```

2. Specify your objectives. For example, if your objectives are execution efficiency and traceability, in that priority, enter:

   ```
   set_param(model_name, 'ObjectivePriorities',...
   {'Execution efficiency', 'Traceability'});
   ```

3. Execute the Code Generation Advisor, by using either the Code Generation Advisor or in the Configuration Parameters dialog box. For more information, see “Configure Model for Code Generation Objectives by Using Code Generation Advisor” on page 29-2.

When you specify a GRT-based system target file, you can specify an objective at the command line. If you specify `ROM efficiency`, `RAM efficiency`, `Traceability`, `MISRA C:2012 guidelines`, `Polyspace`, or `Safety precaution`, the build process changes the objective to `Unspecified` because you have specified a value that is invalid when using a GRT-based target.

Related Examples

- “Configure Model for Code Generation Objectives by Using Code Generation Advisor” on page 29-2
- “Create Custom Code Generation Objectives” on page 29-14
- “Recommended Settings Summary for Model Configuration Parameters”
- “Code Generation Advisor Checks” (Simulink Coder)

Check Model and Configuration for Code Generation

You can use the Model Advisor checks to assess model readiness to generate code. To check and configure your model for code generation application objectives such as traceability or debugging, use the Code Generation Advisor.

For information about	See
Model Advisor	"Run Model Checks" (Simulink)
Code Generation Advisor	"Configure Model for Code Generation Objectives by Using Code Generation Advisor" on page 29-2
Checks available with Simulink Coder	"Simulink Coder Checks" (Simulink Coder)
Checks available with Embedded Coder	"Embedded Coder Checks"

To check model `rtwdemo_throttlecntrl` for code efficiency, use the Model Advisor.

1. Open `rtwdemo_throttlecntrl`. Save a copy as `throttlecntrl` in a writable location on your MATLAB path.
2. To start the Model Advisor, select **Analysis** > **Model Advisor** > **Model Advisor**. A dialog box opens showing the model system hierarchy.
3. Click `throttlecntrl` and then click **OK**. The Model Advisor window opens.
4. Expand **By Task** > **Code Generation Efficiency**. To check your model for code generation efficiency, use the checks in the folder. By default, checks that do not trigger an Update Diagram are selected. The checks available for code generation efficiency depend on whether you have a Simulink Coder or Embedded Coder license.
5. In the left pane, select the remaining checks, and then select **Code Generation Efficiency**.
6. In the right pane, select **Show report after run** and click **Run Selected Checks**. The report shows a **Run Summary** that flags check warnings.
7. Review the report. The warnings highlight issues that impact code efficiency. For more information about the report, see "View Model Advisor Reports" (Simulink).

Check Model During Code Generation

To review a model as part of the code generation process, use the Code Generation Advisor .

1. To select and prioritize your code generation objectives, on the **Configuration Parameters > Code Generation** pane, click **Set Objectives**.
2. On the **Configuration Parameters > Code Generation > General** pane, select one of the following from **Check model before generating code**:
   - `On (proceed with warnings)`
   - `On (stop for warnings)`
3. If you want to only generate code, select **Generate code only**. Otherwise clear the check box to build an executable.
4. Apply your changes. In the model window, press **Ctrl+B** to generate code or build the model.

   If the Code Generation Advisor issues failures or warnings, and you specified:

   - `On (proceed with warnings)` — The Code Generation Advisor window opens while the build process proceeds. After the build process is complete, you can review the results.
   - `On (stop for warnings)` — The build process halts and displays the Diagnostic Viewer. To continue, you must review and resolve the Code Generation Advisor results or clear the **Check model before generating code** parameter.
5. In the Code Generation Advisor window, review the results by selecting a check from the left pane. The results for that check display in the right pane.
6. After reviewing the check results, you can choose to fix warnings and failures as described in "Fix a Model Check Warning or Failure" (Simulink).

Note: When you specify an efficiency or Safety precaution objective, the Code Generation Advisor includes additional checks. When you make changes to one of these additional checks, previous check results can potentially become invalid and need to be rerun.

For more information, see "Set Objectives — Code Generation Advisor Dialog Box" (Simulink Coder)

Embedded Coder Model Advisor Checks for Standards, Guidelines, and Code Efficiency

To check that your model meets guidelines, standards, and is ready to generate code, you can use the Model Advisor checks available with Embedded Coder.

- To start the Model Advisor, in the model window, select **Analysis** > **Model Advisor** > **Model Advisor**.
- In the Model Advisor window, expand the **By Task** folder. The folder contains Model Advisor checks that you can run to help accomplish the task.

For more information about the Model Advisor, see "Run Model Checks" (Simulink).

The table summarizes the Embedded Coder Model Advisor checks that are available in the **By Task** folders.

By Task folder	Model Advisor checks
Modeling Standards for • **IEC 61508, IEC 62304, ISO 26262, and EN 50128** • **Modeling Standards for MAAB**	"Check for blocks not recommended for C/C++ production code deployment"
Code Generation Efficiency	"Identify lookup table blocks that generate expensive out-of-range checking code" "Check output types of logic blocks" "Identify questionable software environment specifications" "Identify questionable code instrumentation (data I/O)" "Identify blocks that generate expensive fixed-point and saturation code" "Identify blocks that generate expensive rounding code"

By Task folder	Model Advisor checks
	"Identify questionable fixed-point operations"
Modeling Standards for DO-178C/DO-331	"Check for blocks not recommended for C/C++ production code deployment" "Check the hardware implementation" "Identify questionable subsystem settings"
Modeling Guidelines for MISRA C:2012	"Check configuration parameters for MISRA C:2012" "Check for blocks not recommended for MISRA C:2012" "Check for unsupported block names" "Check usage of Assignment blocks" "Check for bitwise operations on signed integers" "Check for recursive function calls" "Check for switch case expressions without a default case" "Check for equality and inequality operations on floating-point values"

Related Examples

- "Select and Run Model Advisor Checks" (Simulink)
- "Simulink Coder Model Advisor Checks for Standards and Code Efficiency" (Simulink Coder)
- "Modeling Guidelines for Model Configuration" on page 2-41

Create Custom Code Generation Objectives

In this section...

The Code Generation Advisor reviews your model based on objectives that you specify. If the predefined efficiency, traceability, Safety precaution, and debugging objectives do not meet your requirements, you can create custom objectives.

To create custom objectives:

- Create an objective and add parameters and checks to this new objective.
- Create an objective based on an existing objective, then add, modify, and remove the parameters and checks within that new objective.

Specify Parameters in Custom Objectives

When you create a custom objective, you specify the values of configuration parameters that the Code Generation Advisor reviews. You can use the following methods:

- addParam — Add parameters and specify the values that the Code Generation Advisor reviews in **Check model configuration settings against code generation objectives**.
- modifyInheritedParam — Modify inherited parameter values that the Code Generation Advisor reviews in **Check model configuration settings against code generation objectives**.
- removeInheritedParam — Remove inherited parameters from a new objective that is based on an existing objective. When you select multiple objectives, if another selected objective includes this parameter, the Code Generation Advisor reviews the parameter value in **Check model configuration settings against code generation objectives**.

Specify Checks in Custom Objectives

Objectives include the **Check model configuration settings against code generation objectives** check by default. When you create a custom objective, you specify the list of additional checks that are associated with the custom objective. You can use the following methods:

- addCheck — Add checks to the Code Generation Advisor. When you select the custom objective, the Code Generation Advisor displays the check, unless you specify an additional objective with a higher priority that excludes the check.

  For example, add a check to the Code Generation Advisor to include a custom check in the automatic model checking process.
- excludeCheck — Exclude checks from the Code Generation Advisor. When you select multiple objectives, if you specify an additional objective that includes this check as a higher priority objective, the Code Generation Advisor displays this check.

  For example, exclude a check from the Code Generation Advisor when a check takes a long time to process.
- removeInheritedCheck — Remove inherited checks from a new objective that is based on an existing objective. When you select multiple objectives, if another selected objective includes this check, the Code Generation Advisor displays the check.

  For example, remove an inherited check, rather than exclude the check, when the check takes a long time to process, but the check is important for another objective.

Determine Checks and Parameters in Existing Objectives

When you base a new objective on an existing objective, you can determine what checks and parameters the existing objective contains. The Code Generation Advisor contains the list of checks in each objective.

For example, the `Efficiency` objective includes checks that you can see in the Code Generation Advisor.

1. Open the rtwdemo_rtwecintro model.
2. Specify an ERT-based target.
3. On the model toolbar, select **Code > C/C++ Code > Code Generation Advisor**.
4. In the System Selector window, select the model or subsystem that you want to review, and then click **OK**.

5 In the Code Generation Advisor, on the **Code Generation Objectives** pane, select the code generation objectives. As you select objectives, on the left pane, the Code Generation Advisor updates the list of checks it runs on your model. For this example, select `Execution efficiency`. In **Available objectives**, double-click `Execution efficiency`. `Execution efficiency` is added to **Selected objectives - prioritized**.

In the left pane, the Code Generation Advisor lists the checks for the `Execution efficiency` objective. The first check, **Check model configuration settings against code generation objectives**, lists parameters and values specified by the objective. For example, the Code Generation Advisor displays the list of parameters and the recommended values in the `Execution efficiency` objective. To see the list of parameters and values:

1 Run **Check model configuration settings against code generation objectives**.

2 Click **Modify Parameters**.

3 Rerun the check.

In the check results, the Code Generation Advisor displays the list of parameters and recommended values for the `Execution efficiency` objective.

Passed
Current Objectives: Execution efficiency

The following parameters have been checked and confirmed with the recommended value

Parameter	Value
non-inlined S-functions	off
Suppress error status in real-time model data structure	on
MAT-file logging	off
Classic call interface	off
continuous time	off
non-finite numbers	off
Single output/update function	on
Minimize algebraic loop occurrences	off

Steps to Create Custom Objectives

To create a custom objective:

1 Create an `sl_customization.m` file.

- Specify custom objectives in a single `sl_customization.m` file only or the software generates an error. This issue is true even if you have more than one `sl_customization.m` file on your MATLAB path.

- Except for the *matlabroot*`/work` folder, do not place an `sl_customization.m` file in your root MATLAB folder or its subfolders. Otherwise, the software ignores the customizations that the file specifies.

2 Create an `sl_customization` function that takes a single argument. When the software invokes the function, the value of this argument is the Simulink customization manager. In the function:

- To create a handle to the code generation objective, use the `ObjectiveCustomizer` constructor.
- To register a callback function for the custom objectives, use the `ObjectiveCustomizer.addCallbackObjFcn` method.
- To add a call to execute the callback function, use the `ObjectiveCustomizer.callbackFcn` method.

For example:

```
function sl_customization(cm)
%SL_CUSTOMIZATION objective customization callback

objCustomizer = cm.ObjectiveCustomizer;
index = objCustomizer.addCallbackObjFcn(@addObjectives);
objCustomizer.callbackFcn{index}();

end
```

3 Create a MATLAB callback function that:

- Creates code generation objective objects by using the rtw.codegenObjectives.Objective constructor.
- Adds, modifies, and removes configuration parameters for each objective by using the addParam, modifyInheritedParam, and removeInheritedParam methods.
- Includes and excludes checks for each objective by using the addCheck, excludeCheck, and removeInheritedCheck methods.
- Registers objectives by using the register method.

The following example shows how to create an objective `Reduce RAM Example`. `Reduce RAM Example` includes five parameters and three checks that the Code Generation Advisor reviews.

```
function addObjectives

% Create the custom objective
obj = rtw.codegenObjectives.Objective('ex_ram_1');
```

```
setObjectiveName(obj, 'Reduce RAM Example');

% Add parameters to the objective
addParam(obj, 'DefaultParameterBehavior', 'Inlined');
addParam(obj, 'BooleanDataType', 'on');
addParam(obj, 'OptimizeBlockIOStorage', 'on');
addParam(obj, 'EnhancedBackFolding', 'on');
addParam(obj, 'BooleansAsBitfields', 'on');

% Add additional checks to the objective
% The Code Generation Advisor automatically includes 'Check model
% configuration settings against code generation objectives' in every
% objective.
addCheck(obj, 'mathworks.design.UnconnectedLinesPorts');
addCheck(obj, 'mathworks.design.Update');

%Register the objective
register(obj);

end
```

The following example shows you how to create an objective `My Traceability Example` based on the existing Traceability objective. The custom objective modifies, removes, and adds parameters that the Code Generation Advisor reviews. It also adds and removes checks from the Code Generation Advisor.

```
function addObjectives

% Create the custom objective from an existing objective
obj = rtw.codegenObjectives.Objective('ex_my_trace_1', 'Traceability');
setObjectiveName(obj, 'My Traceability Example');

% Modify parameters in the objective
modifyInheritedParam(obj, 'GenerateTraceReportSf', 'Off');
removeInheritedParam(obj, 'ConditionallyExecuteInputs');
addParam(obj, 'MatFileLogging', 'On');

% Modify checks in the objective
addCheck(obj, 'mathworks.codegen.SWEnvironmentSpec');
removeInheritedCheck(obj, 'mathworks.codegen.CodeInstrumentation');

%Register the objective
register(obj);

end
```

4. If you previously opened the Code Generation Advisor, close the model from which you opened the Code Generation Advisor.
5. Refresh the customization manager. At the MATLAB command line, enter `sl_refresh_customizations`.
6. Open your model and review the new objectives.

Related Examples

- "Configure Model for Code Generation Objectives by Using Code Generation Advisor" on page 29-2
- "Configure Code Generation Objectives Programmatically" on page 29-9
- "Recommended Settings Summary for Model Configuration Parameters"
- "Code Generation Advisor Checks" (Simulink Coder)

Configuration Variations

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 a time. For more information on configuration sets and how to view and edit them in the Configuration Parameters dialog box, see “About Model Configurations” (Simulink).

A configuration set includes parameters that specify code generation in general. For more information, see “Configure a Model for Code Generation” (Simulink Coder). With Embedded Coder and an ERT system target file, more parameters are available for fine-tuning to optimize and customize the appearance of the generated code.

Multiple configuration sets can be 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 can specify the default ERT target with external mode support enabled for rapid prototyping. Another configuration set can specify the ERT-based target for Visual C++® to generate production code for deployment of the application. Activation of either configuration set fully reconfigures the model for that type of code generation.

Related Examples

- “About Model Configurations” (Simulink)
- “Configure a Model for Code Generation” (Simulink Coder)

Configure and Optimize Model with Configuration Wizard Blocks

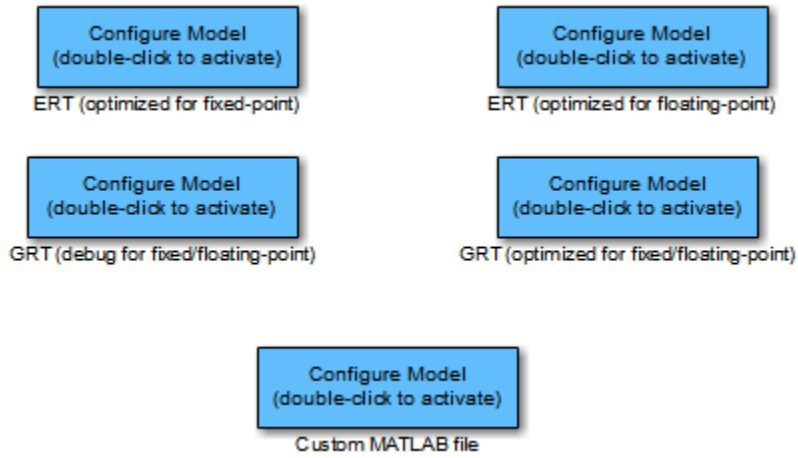
The Embedded Coder software provides a library of Configuration Wizard blocks and scripts to help you configure and optimize code generation from your models.

In this section...

Configuration Wizard Block Library

The library provides a Configuration Wizard block that you can customize. It also provides four preset Configuration Wizard blocks that update the active configuration parameters for a specified goal.

Block	Description
Custom MATLAB file	Update active configuration parameters of parent model by using a custom file
ERT (optimized for fixed-point)	Update active configuration parameters of parent model for ERT fixed-point code generation
ERT (optimized for floating-point)	Update active configuration parameters of parent model for ERT floating-point code generation
GRT (debug for fixed/floating-point)	Update active configuration parameters of parent model for GRT fixed- or floating-point code generation with debugging enabled
GRT (optimized for fixed/floating-point)	Update active configuration parameters of parent model for GRT fixed- or floating-point code generation



When you add one of the preset Configuration Wizard blocks to your model and double-click it, a predefined MATLAB file script configures parameters of the active configuration set without manual intervention. The preset blocks optimally configure the parameters for one of the following cases:

- Fixed-point code generation with the ERT target
- Floating-point code generation with the ERT target
- Fixed-point or floating-point code generation with TLC debugging parameters enabled, with the GRT target.
- Fixed-point or floating-point code generation with the GRT target

The Custom block provides an example MATLAB file script that you can adapt to your requirements.

You can also set up the Configuration Wizard blocks to invoke the build process after configuring the model.

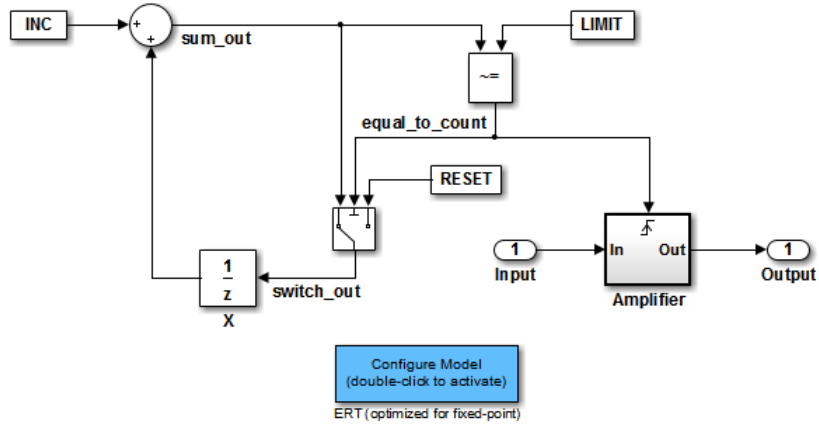
Add a Configuration Wizard Block

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

1. Open the model that you want to configure.
2. Open the Embedded Coder block library by typing the command `rtweclib`.

3 Double-click the Configuration Wizards icon. The Configuration Wizards sublibrary opens.

4 Select the Configuration Wizard block that you want to use and drag it into your model. This model contains the `ERT (optimized for fixed-point)` Configuration Wizard block.



5 If you want the Configuration Wizard block to invoke the build process after configuration, right-click the Configuration Wizard block in your model, and select **Mask > Mask Parameters** from the context menu. Then, select the **Invoke build process after configuration** parameter. Do not change the **Configure the model for** block parameter, unless you want to create a custom block and script. In that case, see "Create a Custom Configuration Wizard Block" on page 29-24.

6 Click **Apply** and close the Mask Parameters dialog box.

7 Save the model.

Use Configuration Wizard Blocks to Configure Your Model

After you add a Configuration Wizard block to your model, to configure your model, double-click the block. The script associated with the block sets parameters of the active configuration set that are relevant to code generation (including selection of the target). You can see that the parameters have changed by opening the Configuration Parameters dialog box and examining the parameter settings.

If you selected the **Invoke build process after configuration** block parameter, the script also initiates the code generation and build process.

Note: To provide a quick way to switch between configurations, you can add more than one Configuration Wizard block to your model.

Create a Custom Configuration Wizard Block

The Custom Configuration Wizard block and the associated MATLAB file script, *matlabroot*/toolbox/rtw/rtw/rtwsampleconfig.m, provide a starting point for customization.

Set Up a Configuration Wizard Block

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. To store your custom script, create a folder. This folder must not be anywhere inside the MATLAB folder structure (that is, it must not be under *matlabroot*).

   The example refers to this folder as /my_wizards.
2. Add the folder to the MATLAB path. Save the path for future sessions.
3. Copy the example script rtwsampleconfig.m in the folder *matlabroot*/toolbox/rtw/rtw (open) to the /my_wizards folder that you created. Then, rename the script. This example 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:

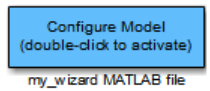
   ```
   disp('Custom Configuration Wizard Script completed.');
   ```

   This statement is used later as a test to see that your custom block has executed the script.
5. Save your script and close the MATLAB editor.

The next task is to create a Simulink library and add a custom block to it.

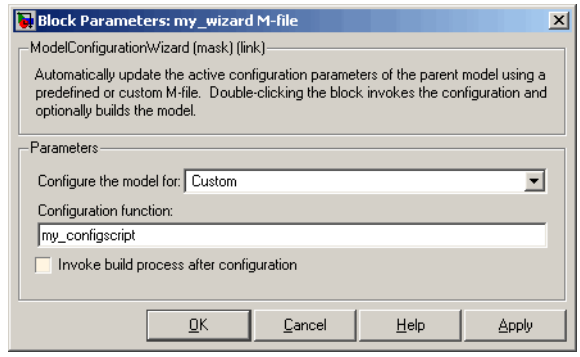
1. Open the Embedded Coder block library and the Configuration Wizards sublibrary, as described in “Add a Configuration Wizard Block” on page 29-22.
2. Select **New > Library** from the **File** menu of the Configuration Wizards sublibrary window. An empty library window opens.

3. Select the Custom MATLAB file block from the Configuration Wizards sublibrary and drag it into the empty library window.
4. To distinguish your custom block from the original, edit the Custom MATLAB file label under the block.
5. Select **Save as** from the **File** menu of the new library window. Save the library to the `/my_wizards` folder, under your library name of choice. In this figure, the library is saved as `ex_custom_button` and the block is labeled `my_wizard MATLAB-file`.



The next task is to link the custom block to the custom script:

1. Right-click the block in your model and select **Mask > Mask Parameters** from the context menu. The **Configure the model for** menu is set to `Custom`. When `Custom` is selected, the **Configuration function** edit field is enabled so that you can enter the name of a custom script.
2. In the **Configuration function** field, enter the name of your custom script . (Do not enter the `.m` file name extension, which is implicit.)



3. By default, the **Invoke build process after configuration** parameter is cleared. You can change the default for your custom block by selecting this option. For now, leave this parameter cleared.
4. Click **Apply** and close the Mask Parameters dialog box.
5. Save the library.

6 Close the Embedded Coder block library and the Configuration Wizards sublibrary. Leave your custom library open for use in the next task.

Test your block and script in a model.

1 Open the `vdp` model by typing the command:

```
vdp
```

2 Open the Configuration Parameters dialog box and view the parameters by clicking **Code Generation** in the list in the left pane of the dialog box.

3 Observe that `vdp` is configured, by default, for the GRT target. Close the Configuration Parameters dialog box.

4 Select your custom block from your custom library. Drag the block into the `vdp` model.

5 In the `vdp` model, double-click your custom block.

6 In the MATLAB window, you see the test message that you previously added to your script:

```
Custom Configuration Wizard Script completed.
```

The test message indicates that the custom block executed the script.

7 Reopen the Configuration Parameters dialog box and view the **Code Generation** pane again. The model is now 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.

Create a Configuration Wizard Script

Create your custom Configuration Wizard script by copying and modifying the example script, `rtwsampleconfig.m`.

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 active configuration set. The Simulink software obtains this handle and passes it in to the configuration function when you double-click a Configuration Wizard block.

Your custom script must conform to this prototype. Your code must use `cs` as a "black-box" object that transmits information to and from the active configuration set.

Access Configuration Set Parameters

To set parameters or obtain parameter values, use the Simulink `set_param` and `get_param` functions.

Option names are passed in to `set_param` and `get_param` as character vectors specifying an *internal option name*. The internal option name can be different from the option label on the UI (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 UI option labels. For example:

```
set_param(cs,'LifeSpan','1'); % Application lifespan (days)
```

To obtain the current setting of an option in the active configuration set, call `get_param`. Pass in the `cs` object as the first argument, followed by the internal option name. For example, the following code excerpt tests the setting of the **Create code generation report** option:

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

To set an option in the active configuration set, call `set_param`. Pass in the `cs` object as the first argument, followed by one or more parameter/value pairs that specify the internal option name and its value. For example, the following code excerpt turns off the **Support absolute time** option:

```
set_param(cs,'SupportAbsoluteTime','off');
```

Select 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 character vector 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';
```

You select the system target file by passing the `cs` object and the `stf` character vector to the `switchTarget` function:

```
switchTarget(cs,stf,[]);
```

Set the template makefile and make command options by using `set_param` calls:

```
set_param(cs,'TemplateMakefile',tmf);
set_param(cs,'MakeCommand',mc);
```

To select a target, your custom script must set up the character vector variables `stf`, `tmf`, and `mc` and pass them to the calls.

Obtain 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 the currently selected target is derived from the ERT target by checking the `IsERTTarget` property, as follows:

  ```
  isERT = strcmp(get_param(cs,'IsERTTarget'),'on');
  ```

You can use this information to determine whether the script must configure ERT-specific parameters, 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
```

Invoke a Configuration Wizard Script from the Command Line

Configuration Wizard scripts can be run from the MATLAB command line.

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');
```

Related Examples

- “Generate Code with the Quick Start Tool” on page 34-10
- “Generate Code and Simulate Models in a Simulink Project”
- “Generate Code and Simulate Models with Simulink Project API”

Create a Model Configured for Code Generation Using Model Templates

Model templates provide you with a starting point for quickly developing models for code generation. Embedded Coder templates provide starting models for the following applications:

- Code Generation System. Create a model to get started with code generation.
- Exported functions. Create a model for generating code from function-call subsystems.
- Fixed-step, multirate. Create a fixed-step model with multiple rates for production code generation.
- Fixed-step, single-rate. Create a fixed-step model with a single rate for production code generation.

In the templates, traceability and reporting are turned on so that you can easily evaluate your generated code. The model has **System target file** set to `ert.tlc` and is configured to meet code generation objectives prioritized in the following order:

1. Execution efficiency
2. Traceability

To create a model using a model template:

1. On the MATLAB home tab, click **Simulink**.
2. In the Simulink start page, expand **Embedded Coder**.
3. Select a template.
4. Click **Create**. A new model that uses the template contents and settings appears in the Simulink Editor window.

For more information, for example to create and use a template as a reference design, see "Create a Template from a Model" (Simulink).

System Target File Configuration

Select a System Target File

To configure a model for code generation, follow the steps in "Select a Solver That Supports Code Generation" (Simulink Coder) and "Select a System Target File from STF Browser" (Simulink Coder). When you select a system target file, other model configuration parameters change to serve requirements of the execution environment. For example:

- Code interface parameters
- Build process parameters, such as the toolchain or template makefile
- Target hardware parameters, such as word size and byte ordering

After selecting a system target file, you can modify model configuration parameter settings.

You can switch between different system target files in a single workflow for different code generation purposes (for example, rapid prototyping versus product code deployment). To switch, set up different configuration sets for the same model and switch the active configuration set for the current operation. For more information on how to set up configuration sets and change the active configuration set, see "Manage a Configuration Set" (Simulink).

In this section...

Select a Solver That Supports Code Generation

To build a model, the model configuration must select a solver that is compatible with code generation for the system target file. Few system target files support code generation with variable-step solvers or for models with a nonzero start time.

- Use **Configuration Parameters** > **Solver** > **Type** and select `Fixed-step` for GRT, ERT, and ERT-based system target files.
- Use **Configuration Parameters** > **Solver** > **Type** and select `Fixed-step` or `Variable-step` for Rapid Simulation (Rsim) or S-Function (rtwsfcn) system target files.

For more information, see “Time-Based Scheduling and Code Generation” on page 16-2.

When you try to build models with a nonzero start time using a system target file does not support a nonzero start time, the code generator does not produce code. The build process displays an error message. The Rapid Simulation (RSim) system target file supports a nonzero start time when **Configuration Parameters > RSim Target > Solver selection** is set to `Use Simulink solver module`. Other system target files do not support a nonzero start time.

Select a System Target File from STF Browser

After you follow the steps in “Select a Solver That Supports Code Generation” (Simulink Coder), use **Configuration Parameters > Code Generation > System target file** and click the **Browse** button to open the System Target File Browser. Select a system target file from the list. Your selection appears in the **System target file** field (*`target`*`.tlc`).

If you use a system target file that does not appear in the System Target File Browser, enter the name of your system target file in the **System target file** field.

You also can select a system target file programmatically from MATLAB code, as described in “Select a System Target File Programmatically” on page 30-4.

After selecting a system target file, you can modify model configuration parameter settings. Selecting a system target file for your model selects either the toolchain approach or template makefile approach for build process control. For more information about these approaches, see “Choose and Configure Build Process” on page 40-14.

If you want to switch between different system target files in a single workflow for different code generation purposes, set up different configuration sets for the same model. Switch the active configuration set for the current operation. This approach is useful for switching between rapid prototyping and product code deployment. For more information on how to set up configuration sets and change the active configuration set, see “Manage a Configuration Set” (Simulink).

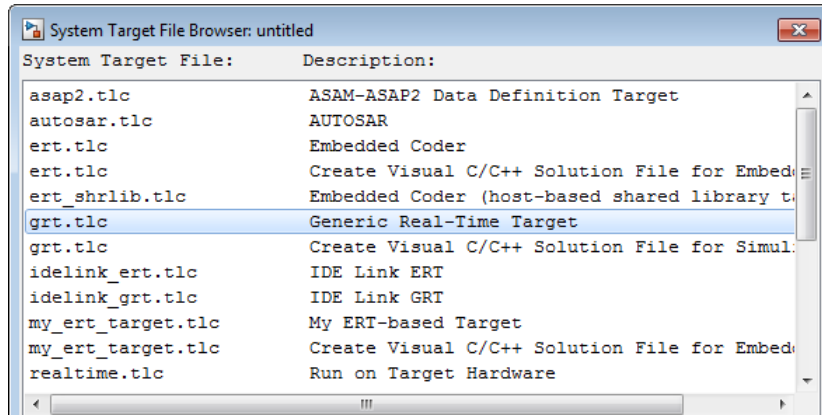
To select a system target file using the System Target File Browser,

1. Open the **Code Generation** pane of the Configuration Parameters dialog box.
2. Click the **Browse** button next to the **System target file** field. This button opens the System Target File Browser. The browser displays a list of currently available

system target files, including customizations. When you select a system target file, the code generator automatically chooses the system target file, toolchain or template makefile, and/or `make` command for that configuration.

The next step shows the System Target File Browser with the GRT system target file selected.

3 Click the desired entry in the list of available configurations. The background of the list box turns yellow to indicate that an unapplied choice has been made. To apply it, click **Apply** or **OK**.



System Target File Browser

When you choose a system target file, the code generator selects the toolchain or template makefile and/or `make` command for that configuration and displays them in the **System target file** field. The description of the system target file from the browser is placed below its name in the **Code Generation** pane. For information each system target file, see “Compare System Target File Support” on page 30-21.

Select a System Target File Programmatically

Simulink models store model-wide parameters and system target file-specific data in *configuration sets*. Every configuration set contains a component that defines the structure of a particular system target file and the current values of relevant options. Simulink loads some of this information from the system target file that you specify. You can configure models to generate alternative code by copying and modifying old or adding

new configuration sets and browsing to select a new system target file. Then, you can interactively select an active configuration from among these sets (only one configuration set can be active at a given time).

Scripts that automate system target file selection must emulate this process.

To program system target file selection:

1. Obtain a handle to the active configuration set with a call to the `getActiveConfigSet` function.
2. Define character vector variables that correspond to the required system target file, toolchain or template makefile, and/or `make` command settings. For example, for the ERT system target file, you would define variables for the character vectors `'ert.tlc'`, `'ert_default_tmf'`, and `'make_rtw'`.
3. Select the system target file with a call to the `switchTarget` function. In the function call, specify the handle for the active configuration set and the system target file.
4. Set the `TemplateMakefile` and `MakeCommand` configuration parameters to the corresponding variables created in step 2.

For example:

```
cs = getActiveConfigSet(model);
stf = 'ert.tlc';
tmf = 'ert_default_tmf';
mc  = 'make_rtw';
switchTarget(cs,stf,[]);
set_param(cs,'TemplateMakefile',tmf);
set_param(cs,'MakeCommand',mc);
```

For more information about selecting system target files programmatically, see `switchTarget`.

Develop Custom System Target Files

You can create your own system target files that interface with external code or operating environments.

For more information on how to make custom system target files appear in the System Target File Browser and display relevant controls, see “About Embedded Target Development” (Simulink Coder) and the topics it references.

See Also

`getActiveConfigSet` | `switchTarget`

More About

- “Configure STF-Related Code Generation Parameters” on page 30-7
- “Configure a Code Replacement Library” on page 30-17
- “Configure Standard Math Library for Target System” on page 30-18
- “Compare System Target File Support” on page 30-21
- “About Embedded Target Development” (Simulink Coder)

Configure STF-Related Code Generation Parameters

Many model configuration parameters for code generation are specific to GRT, ERT, or ERT-based system target files. For more information, see the following topics.

In this section...

Specify Generated Code Interfaces

Use interface model configuration parameters to control which libraries to use when generating code, whether to include support for an API in generated code, and other interface options.

To...	Select or Enter...
Specify the standard math library that the code generator uses when generating code.	Select `C89/C90 (ANSI)`, `C99 (ISO)`, or `C++03 (ISO)` for the **Standard math library** parameter on the **All Parameters** tab. Selecting `C89/C90 (ANSI)` provides the ANSI[a] C set of library functions. For example, selecting `C89/C90 (ANSI)` results in generated code that calls `sin()` whether the input argument is double precision or single precision. However, if you select `C99 (ISO)`, the generated code calls the function `sinf()` when the input argument is single precision. If your compiler supports the ISO®[b] C math extensions, selecting the ISO C library can result in more efficient code. For more information, see "Standard math library" (Simulink Coder). The options for this parameter have dependencies. See Interface Dependencies.

To...	Select or Enter...
Specify an application-specific library that the code generator uses when generating code.	If you generate application-specific C or C++ code for math functions or operations, select a value for **Code replacement library**. Otherwise, specify `None`. For more information about code replacement libraries, see "Choose a Code Replacement Library" on page 37-9 and "Code replacement library" (Simulink Coder). The options for this parameter have dependencies. See Interface Dependencies.
Direct where the code generator places fixed-point and other utility code.	Select `Auto` or `Shared location` for **Shared code placement**. The shared location directs code for utilities to be placed within the `slprj` folder in your working folder, which is used for building referenced models. If you select `Auto`, • When the model contains Model blocks, places utility code within the `slprj/`*`target`*`/_sharedutils` folder. • When the model does not contain Model blocks, places utility code in the build folder (generally, in *`model`*`.c` or *`model`*`.cpp`).
Specify text to be added to the variable names used when logging data to MAT-files and to distinguish logging data from code generation and simulation applications.	Enter a prefix or suffix, such as `rt_` or `_rt`, for the **MAT-file variable name modifier** parameter on the **All Parameters** tab. The code generator prefixes or appends the text to the variable names for system outputs, states, and simulation time specified in the **Data Import/Export** pane. See "Log Program Execution Results" on page 42-2 for information on MAT-file data logging.

To...	Select or Enter...
Specify data exchange APIs to be included in generated code.	Select one or more **C API** options, the **ASAP2 interface** option, or the **External mode** option. When you select **External mode**, other options appear. The data exchange APIs are independent, and you can select combinations of these APIs. For example, you could choose C API and external mode. For more information on working with these interfaces, see "Exchange Data Between Generated and External Code Using C API" on page 43-2, "Export ASAP2 File for Data Measurement and Calibration" on page 44-2, and "What You Can Do with a Host/ Target Communication Channel" on page 41-2. The options for this parameter have dependencies. See Interface Dependencies.

a. ANSI is a registered trademark of the American National Standards Institute, Inc.
b. ISO is a registered trademark of the International Organization for Standardization.

Note: Before setting **Standard math library** or **Code replacement library**, verify that your compiler supports the library you want to use. If you select a parameter value that your toolchain does not support, compiler errors can occur. For example, if you select standard math library `C99 (ISO)` and your compiler does not support the ISO C math extensions, compile-time errors could occur.

When the Embedded Coder product is installed on your system, the **Code Generation** > **Interface** pane expands to include several additional options. For descriptions of **Code Generation** > **Interface** pane parameters, see "Model Configuration Parameters: Code Generation Interface" (Simulink Coder).

Several interface parameters have dependencies on settings of other parameters. The following table summarizes the dependencies.

Interface Dependencies

Parameter	Dependencies?	Dependency Details
Standard math library	Yes	Available values depend on **Language** selection.

Parameter	Dependencies?	Dependency Details
Code replacement library	Yes	Available values depend on product licensing and other parameters. For more information, see "Code replacement library" (Simulink Coder).
Shared code placement	No	
Support: floating-point numbers (ERT system target files only)	No	
Support: non-finite numbers	Yes (ERT) No (GRT)	For ERT system target files, enabled by **Support floating-point numbers**
Support: complex numbers (ERT system target files only)	No	
Support: absolute time (ERT system target files only)	No	
Support: continuous time (ERT system target files only)	Yes	Requires that you disable **Remove error status field in real-time model data structure**.
Support: non-inlined S-functions (ERT system target files only)	Yes	Requires that you enable **Support floating-point numbers** and **Support non-finite numbers**
Classic call interface	Yes	Requires that you disable **Single output/update function**. For ERT system target files, requires that you enable **Support floating-point numbers**.
Single output/update function	Yes	Disable for **Classic call interface**
Terminate function required (ERT system target files only)	Yes	
Code interface packaging	Yes	Available values depend on **Language** selection.
Multi-instance code error diagnostic	Yes	Set **Code interface packaging** to `Reusable function` or `C++ class`

Parameter	Dependencies?	Dependency Details
Pass root-level I/O as (ERT system target files only)	Yes	Set **Code interface packaging** to `Reusable function`
Use dynamic memory allocation for model initialization (ERT system target files only)	Yes	Set **Code interface packaging** to `Reusable function`
MAT-file logging	Yes	For GRT system target files, requires that you enable **Support non-finite numbers**; for ERT system target files, requires that you enable **Support floating-point numbers**, **Support non-finite numbers**, and **Terminate function required**
MAT-file file variable name modifier	Yes	Enabled by **MAT-file logging**
Remove error status field in real-time model data structure (ERT system target files only)	Yes	Requires that you disable **Support: continuous time**.
Generate C API for: signals	No	
Generate C API for: parameters	No	
Generate C API for: states	No	
Generate C API for: root-level I/O	No	
ASAP2 interface	No	
External mode	No	
Transport layer	Yes	Enable **External mode**
MEX-file arguments	Yes	Enable **External mode**
Static memory allocation	Yes	Enable **External mode**
Static memory buffer size	Yes	Enable **Static memory allocation**

Configure Numeric Data Support

By default, ERT system target files support code generation for integer, floating-point, nonfinite, and complex numbers.

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

For more information, see "Model Configuration Parameters: Code Generation Interface" (Simulink Coder).

Configure Time Value Support

Certain blocks require the value of absolute time, elapsed time, or continuous time. Absolute time is the time from the start of program execution to the present time. Elapsed time is the time elapsed between two trigger events. Depending on the blocks used, your model could require adjustment of 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 "Absolute and Elapsed Time Computation" (Simulink Coder). If you do not select this parameter and the model includes a block that uses 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 "Model Configuration Parameters: Code Generation Interface" (Simulink Coder).

Configure Noninlined S-Function Support

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

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

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, clear **Support non-inlined S-functions**.

When generating code for a model that contains noninlined S-functions with an ERT system target file, there could be a mismatch between the simulation and code generation results when either of the following is true:

- Model configuration parameter `GenCodeOnly` is set to `off` or **Configuration Parameters** > **Code Generation** > **Generate code only** is cleared.
- Model configuration parameter `ProdEqTarget` is set to `off`.

To avoid such a mismatch, set `ProdEqTarget` to `on` or select **Configuration Parameters** > **Code Generation** > **Generate code only** (or set `GenCodeOnly` to `on`).

Configure Model Function Generation and Argument Passing

For ERT system target files, you can configure model for how 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 pre-R2012a GRT system target file (`grt_main.c` or `.cpp`).	Select **Classic call interface** and **MAT-file logging**. In addition, clearing **Remove error status field in real-time model data structure**. **Classic call interface** provides a quick way to use code generated in R2012a or higher with a pre-R2012a GRT-based custom system target file by generating wrapper function calls that interface to the generated code.

To...	Do...
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 the model configuration enables "Single output/update function" (Simulink Coder) (the default). See "Model Blocks and Direct Feed through" (Simulink).
Generate a *model*_terminate function for a model not designed to run indefinitely.	Select **Terminate function required** (Simulink Coder). For more information, see the description of model_terminate.
Generate reusable, reentrant code from a model or subsystem.	Select **Generate reusable code**. See "Configure Code Reuse Support" on page 30-15 for details.
Statically allocate model data structures and access them directly in the model code.	Clear **Generate reusable code**. The generated code is not reusable or reentrant. See "Entry-Point Functions and Scheduling" (Simulink Coder) 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 require to log or monitor error messages.	Select **Remove error status field in real-time model data structure**. Selecting this parameter can also cause the code generator to omit the rtModel structure 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 could exhibit unexpected behavior. For example, if you select the option in one model but not in another, it is possible that the integrated application could not register the error status. Do not select this parameter if you select the **MAT-file logging** option. The two options are incompatible.

To...	Do...
Open the Model Step Functions dialog box preview and modify the *model*_step function prototype (see "Entry-Point Functions and Scheduling" (Simulink Coder)).	Click **Configure Step Function**. 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 Step Function** button and the Model Step Functions dialog box, see "Control Generation of Function Prototypes" on page 26-2.

For more information, see "Model Configuration Parameters: Code Generation Interface" (Simulink Coder).

Configure Code Reuse Support

For GRT, ERT, GRT-based, and ERT-based system target files, you can configure how a model reuses code by setting the **Configuration Parameters** > **Code Generation** > **Code interface packaging** parameter value to `Reusable function`.

The **Configuration Parameters** > **Code Generation** > **Pass root-level I/O as** parameter 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)	**Code interface packaging** > `Reusable function` 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	**Code interface packaging** > `Reusable function` and **Pass root-level I/O as** > `Structure reference`.
Pack root-level input arguments and root-level output arguments into the model data structure to support reentrant multi-instance code from a model for ERT system target file	**Code interface packaging** > `Reusable function` and **Pass root-level I/O as** > `Part of model data structure`.

If using the **Code interface packaging** > `Reusable function` selection, consider using the **Use dynamic memory allocation for model initialization** option to control

whether an allocation function is generated. This option applies for ERT system target files.

Sometimes, selecting **Code interface packaging** as `Reusable function` can generate code that compiles but is not reentrant. For example, if a signal, `DWork` structure, or parameter data has a storage class other than `Auto`, global data structures are generated. To handle such cases, use the **Multi-instance code error diagnostic** parameter to choose the severity levels for diagnostics.

Sometimes, the code generator is unable to generate valid and compilable code. For example, if the model contains one of the following, the generated code is 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 "Generate Reentrant Code from Top-Level Models" on page 34-20 and "Model Configuration Parameters: Code Generation Interface" (Simulink Coder).

More About

- "Select a System Target File" on page 30-2
- "Configure a Code Replacement Library" on page 30-17
- "Configure Standard Math Library for Target System" on page 30-18
- "Compare System Target File Support" on page 30-21

Configure a Code Replacement Library

You can configure the code generator to change the code that it generates for functions and operators such that the code meets application requirements. Configure the code generator to apply a code replacement library (CRL) during code generation. If you have Embedded Coder, you can develop and apply custom code replacement libraries.

For more information about replacing code, using code replacement libraries that MathWorks provides, see "What Is Code Replacement?" on page 38-2 and "Code Replacement Libraries" (Simulink Coder). For information about developing code replacement libraries, see "What Is Code Replacement Customization?" on page 51-3.

More About

- "Select a System Target File" on page 30-2
- "Configure STF-Related Code Generation Parameters" on page 30-7
- "Configure Standard Math Library for Target System" on page 30-18
- "Compare System Target File Support" on page 30-21
- "Specify Generated Code Interfaces" on page 30-7

Configure Standard Math Library for Target System

Specify standard library extensions that the code generator uses for math operations. When you generate code for a new model or with a new configuration set object, the code generator uses the ISO®/IEC 9899:1999 C (C99 (ISO)) library by default. For preexisting models and configuration set objects, the code generator uses the library specified by the **Standard math library** parameter.

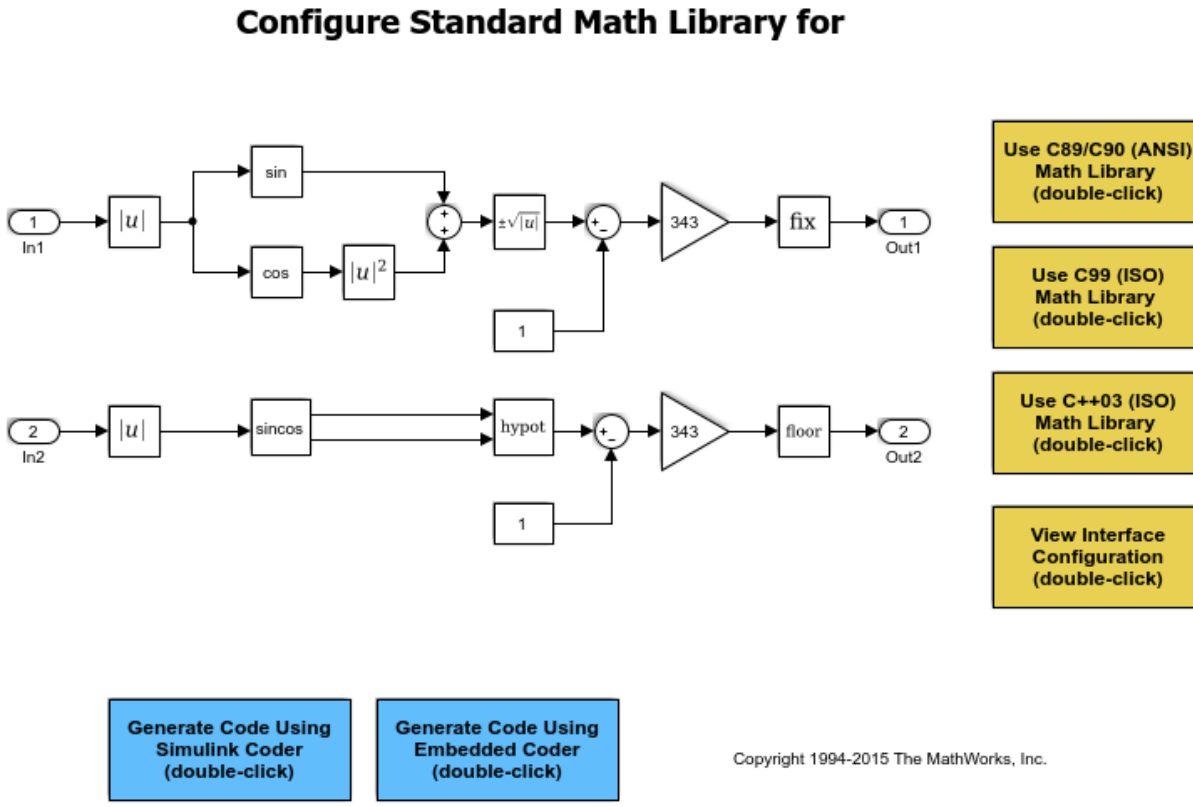
If your compiler supports the ISO®/IEC 9899:1990 (C89/C90 (ANSI)) or ISO/IEC 14882:2003(C++03 (ISO)) math library extensions, you can change the standard math library setting. The C++03 (ISO) library is an option when you select C++ for the programming language.

The C99 library leverages the performance that a compiler offers over standard ANSI C. When using the C99 library, the code generator produces calls to ISO C functions when possible. For example, the generated code calls the function `sqrtf()`, which operates on single-precision data, instead of `sqrt()`.

To change the library setting, use the **Configuration Parameters>All Parameters>Standard math library** parameter. The command-line equivalent is `TargetLangStandard`.

Generate and Inspect ANSI C Code

1. Open the example model `rtwdemo_clibsup`.



2. Generate code.

```
### Starting build procedure for model: rtwdemo_clibsup
### Successful completion of code generation for model: rtwdemo_clibsup
```

3. Examine the code in the generated file `rtwdemo_clibsup.c`. Note that the code calls the `sqrt` function.

```
if (rtb_Abs2 < 0.0F) {
  rtb_Abs2 = -(real32_T)sqrt((real32_T)fabs(rtb_Abs2));
} else {
  rtb_Abs2 = (real32_T)sqrt(rtb_Abs2);
}
```

Generate and Inspect ISO C Code

1. Change the setting of **All Parameters>Standard math library** to `C99 (ISO)`. Alternatively, at the command line, set `TargetLangStandard` to `C99 (ISO)`.

2. Regenerate the code.

```
### Starting build procedure for model: rtwdemo_clibsup
### Successful completion of code generation for model: rtwdemo_clibsup
```

3. Rexamine the code in the generated file `rtwdemo_clibsup.c`. Now the generated code calls the function `sqrtf` instead of `sqrt`.

```
if (rtb_Abs2 < 0.0F) {
  rtb_Abs2 = -sqrtf(fabsf(rtb_Abs2));
} else {
  rtb_Abs2 = sqrtf(rtb_Abs2);
}
```

Related Information

- "Standard math library" (Simulink Coder)
- "Select a System Target File" (Simulink Coder)
- "Configure STF-Related Code Generation Parameters" (Simulink Coder)
- "Configure a Code Replacement Library" (Simulink Coder)
- "Compare System Target File Support" (Simulink Coder)
- "Replace Code Generated from Simulink Models" (Simulink Coder)

Compare System Target File Support

A system target file (such as `grt.tlc`) defines a run-time environment. The code generator uses the system target file to produce code intended for execution on certain target hardware or operating system. The system target file invokes other run-time environment-specific files. For more information on configuring model code generation parameters for target hardware, see "Configure Run-Time Environment Options" (Simulink Coder).

Different types of system target files support a selection of generated code features. In the system target file, the value of the `CodeFormat` TLC variable and corresponding `rtwgensettings.DerivedFrom` field value identify the system target file type and generated code features. These selections control decisions made at several points in the code generation process. These decisions include whether and how the model build generates:

- Certain data structures (for example, `SimStruct` or `rtModel`)
- Static or dynamic memory allocation code
- Calling interface for generated model functions

For custom system target file development, follow these guidelines:

- If the system target file does not include a value for the `CodeFormat` TLC variable, the default value is `RealTime` for generic real-time target (GRT). The corresponding `rtwgensettings.DerivedFrom` field value is `grt.tlc` (default value).
- If you are developing a custom system target file and you have a license for Embedded Coder software, consider setting the `CodeFormat` TLC variable value to `Embedded-C` for embedded real-time target (ERT). The corresponding `rtwgensettings.DerivedFrom` field value is `ert.tlc`. The ERT system target file supports more generated code features than the GRT system target file.

The following example shows how the value for the `CodeFormat` TLC variable and corresponding `rtwgensettings.DerivedFrom` field value are set in `ert.tlc`.

```
%assign CodeFormat = "Embedded-C"

/%
  BEGIN_RTW_OPTIONS
  rtwgensettings.DerivedFrom = 'ert.tlc';
  END_RTW_OPTIONS
%/
```

Warning: You must use the value for the `CodeFormat` TLC variable with its corresponding `rtwgensettings.DerivedFrom` field value to generated code for the model. If none are explicitly selected, the default values correspond. For more information, see "System Target File Structure" (Simulink Coder).

For a description of the optimized call interface generated by default for both the GRT and ERT system target files, see "Entry-Point Functions and Scheduling" (Simulink Coder).

The real-time model data structure (`rtModel`) encapsulates model-specific information in a much more compact form than the `SimStruct`. Many efficiencies related to generated code depend on generation of `rtModel` rather than `SimStruct`, including:

- Integer absolute and elapsed timing services
- Independent timers for asynchronous tasks
- Generation of improved C API code for signal, state, and parameter monitoring
- Pruning the data structure to minimize its size (ERT-derived system target files only)

For a description of the `rtModel`data structure, see "Use the Real-Time Model Data Structure" (Simulink Coder).

The following topics provide more information about generated code features:

In this section...
"Evaluate Product System Target Files" on page 30-22
"Compare Code Styles and STF Support" on page 30-25
"Compare Generated Code Features by Product" on page 30-26
"Compare Generated Code Features by STF" on page 30-29

Evaluate Product System Target Files

The following table lists supported system target files.

Note You can select from a range of system target files by using the System Target File Browser. This selection lets you experiment with configuration options and save your model with different configurations. However, you cannot build or generate code for

non-GRT system target files, unless you have the required license on your system. For example, you require Embedded Coder for ERT system target files, require Simulink Desktop Real-Time for SLDRT system target files, and so on.

Selecting a system target file for your model selects either the toolchain approach or template makefile approach for build process control. For more information about these approaches, see “Choose and Configure Build Process” on page 40-14.

System Target Files Available from System Target File Browser

System Target File	File Names	Reference
Embedded Coder (for PC or UNIX[a] platforms)	`ert.tlc` `ert_shrlib.tlc`	"Select a System Target File" on page 30-2
Create Visual C++[b] Solution File for Embedded Coder	`ert.tlc` (Requires `RTW.MSVCBuild` as TMF[c])	"Select a System Target File" on page 30-2
Embedded Coder for AUTOSAR	`autosar.tlc`	"Generate AUTOSAR-Compliant C Code and Export ARXML Descriptions"
Generic Real-Time (for PC or UNIX platforms)	`grt.tlc`	"Compare Generated Code Features by STF" on page 30-29
Create Visual C++ Solution File	`grt.tlc` (Requires `RTW.MSVCBuild` as TMF[c])	"Compare Generated Code Features by STF" on page 30-29
Rapid Simulation (default for PC or UNIX platforms)	`rsim.tlc`	"Accelerate, Refine, and Test Hybrid Dynamic System on Host Computer by Using RSim System Target File" on page 46-2
Rapid Simulation for LCC compiler	`rsim.tlc`	"Accelerate, Refine, and Test Hybrid Dynamic System on Host Computer by Using RSim System Target File" on page 46-2
Rapid Simulation for UNIX platforms	`rsim.tlc`	"Accelerate, Refine, and Test Hybrid Dynamic System on Host Computer by Using RSim System Target File" on page 46-2
Rapid Simulation for Visual C++ compiler	`rsim.tlc`	"Accelerate, Refine, and Test Hybrid Dynamic System on Host Computer by Using RSim System Target File" on page 46-2

System Target File	File Names	Reference
S-Function for PC or UNIX platforms	`rtwsfcn.tlc`	"Accelerate Simulation, Reuse Code, or Protect Intellectual Property by Using S-Function Target" (Simulink Coder)
S-Function for LCC	`rtwsfcn.tlc`	"Accelerate Simulation, Reuse Code, or Protect Intellectual Property by Using S-Function Target" (Simulink Coder)
S-Function for UNIX platforms	`rtwsfcn.tlc`	"Accelerate Simulation, Reuse Code, or Protect Intellectual Property by Using S-Function Target" (Simulink Coder)
S-Function for Visual C++ compiler	`rtwsfcn.tlc`	"Accelerate Simulation, Reuse Code, or Protect Intellectual Property by Using S-Function Target" (Simulink Coder)
ASAM-ASAP2 Data Definition	`asap2.tlc`	"Export ASAP2 File for Data Measurement and Calibration" on page 44-2
Simulink Desktop Real-Time	`sldrt.tlc` `sldrtert.tlc`	"Set External Mode Code Generation Parameters" (Simulink Desktop Real-Time)
Simulink Real-Time	`slrt.tlc`	"Simulink Real-Time Options Pane" (Simulink Real-Time)
IDE Link capability	`idelink_grt.tlc` `idelink_ert.tlc`	Embedded IDE or target topics such as "Model Setup" on page 72-2

a. UNIX is a registered trademark of The Open Group in the United States and other countries.
b. Visual C++ is a registered trademark of Microsoft Corporation.
c. Set RTW.MSVCBuild in the "Template makefile" (Simulink Coder) field. This creates and builds Visual C++ Solution (.sln) file with Debug configuration.

Compare Code Styles and STF Support

The code generator produces two styles of code. One code style is suitable for rapid prototyping (and simulation by using code generation). The other style is suitable for embedded applications. The following table maps system target files to corresponding code styles.

Code Styles Listed by System Target File

System Target File	Code Style	Purpose
Embedded Coder embedded real-time (ERT)	Embedded	A starting point for embedded application development of C/C++ generated code
Simulink Coder generic real-time (GRT)	Rapid prototyping	A starting point for creating a rapid prototyping target hardware that does not use real-time operating system tasking primitives and for verifying the generated C/C++ code on your desktop computer
Rapid simulation (RSim)	Rapid prototyping	Provides non-real-time simulation on your desktop computer and a high-speed or batch simulation tool
S-function	Rapid prototyping	Creates a C MEX S-function for simulation within another Simulink model
Simulink Desktop Real-Time	Rapid prototyping	Runs model in real time at interrupt level while your desktop computer runs Microsoft Windows in the background
Simulink Real-Time	Rapid prototyping	Runs model in real time on a desktop computer running the Simulink Real-Time kernel

Third-party vendors supply additional system target files for the code generator. For more information about third-party products, see the MathWorks Connections program web page: http://www.mathworks.com/products/connections.

Compare Generated Code Features by Product

The code generation process for real-time system target files (such as GRT) provides many embedded code optimizations. GRT and ERT system target files have many common features. But, selecting an ERT-based system target file offers more extensive features. The system target file selection determines the available features for the code generation product. The following table compares code features available with Simulink Coder and features available with Embedded Coder.

Compare Code Generation Features for Simulink Coder Versus Embedded Coder

Feature	Simulink Coder	Embedded Coder
`rtModel` data structure	• Full `rtModel` structure generated • GRT variable declaration: `rtModel_model model_M_;`	• `rtModel` is optimized for the model • Optional suppression of error status field and data logging fields • ERT variable declaration: `RT_MODEL_model model_M_;`
Custom storage classes (CSCs)	Code generation ignores CSCs; objects are assigned a CSC default to `Auto` storage class	Code generation with CSCs is supported
HTML code generation report	Basic HTML code generation report	Enhanced report with additional detail and hyperlinks to the model
Symbol formatting	Symbols (for signals, parameters, and so on) are generated in accordance with hard-coded default	Detailed control over generated symbols.
User-defined maximum identifier length for generated symbols	Supported	Supported
Generation of terminate function	Generated	Option to suppress the terminate function
Combined output/ update function	Separate output/update functions are generated	Option to generate combined output/ update function
Optimized data initialization	Not available	Options to suppress generation of unnecessary initialization code for zero-valued memory, I/O ports, and so on
Comments generation	Basic options to include or suppress comment generation	Options to include Simulink block descriptions, Stateflow object descriptions, and Simulink data object descriptions in comments
Module Packaging Features (MPF)	Not supported	Extensive code customization features (See “What Are User-

Feature	Simulink Coder	Embedded Coder
		Defined Data Types?" on page 21-2 and "Custom Storage Classes".)
System target file-optimized data types header file	Requires full `tmwtypes.h` header file	Generates optimized `rtwtypes.h` header file, including definitions required by the system target file
User-defined types	User-defined types default to base types in code generation	User-defined data type aliases are supported in code generation
Rate grouping	Not supported	Supported
Auto-generation of main program module	Not supported; static main program module is provided.	Automated and customizable generation of main program module is supported (static main program also available)
Reusable (multi-instance) code generation	Option to generate reusable code with dynamic memory allocation	Option to generate reusable code with static or dynamic memory allocation
Software constraint options	Support for floating point, complex, and nonfinite numbers is enabled	Options to enable or disable support for floating-point, complex, and nonfinite numbers
Application life span	Defaults to `inf`	User-specified; determines most efficient word size for integer timers
Software-in-the-loop (SIL) testing	Model reference simulation target can be used for SIL testing	Additional SIL testing support by using auto-generation of SIL block
ANSI[a]-C/C++ code generation	Supported	Supported
ISO[b]-C/C++ code generation	Supported	Supported
GNU®[c]-C/C++ code generation	Supported	Supported
Generate scalar inlined parameters as `#DEFINE` statements	Not supported	Supported
MAT-file variable name modifier	Supported	Supported

Feature	Simulink Coder	Embedded Coder
Data exchange: C API, ASAP2, external mode	Supported	Supported

a. ANSI is a registered trademark of the American National Standards Institute, Inc.
b. ISO is a registered trademark of the International Organization for Standardization.
c. GNU is a registered trademark of the Free Software Foundation.

Compare Generated Code Features by STF

The code generator supports a selection of generated code features for different types of system target files. In each system target file, the value of the `CodeFormat` TLC variable identifies the set of features.

The following table summarizes how different system target files support applications:

Application	System Target File (STF)
Fixed- or variable-step acceleration	RSIM, S-Function, Model Reference
Fixed-step real-time deployment	GRT, ERT, Simulink Real-Time, Simulink Desktop Real-Time, ...

The following table summarizes the various options available for each **System target file** selection, with the exceptions noted.

Features Supported in Code Generated for System Target Files (STF)

	System Target Files (STF)							
Feature	**grt.tlc**[1]	**ert.tlc**[1]	**ert_shrlib.tlc**[1]	**rtwsfcn.tlc**[1]	**rsim.tlc**[1]	**sldrt.tlc**[1]	**slrt.tlc**[1]	**Other**[1]
Static memory allocation	X	X				X	X	X
Dynamic memory allocation	X[4, 5]	X[4, 5]		X	X		X	
Continuous time	X	X		X	X	X	X	
C/C++ MEX S-functions	X	X		X	X	X	X	

	System Target Files (STF)							
Feature	**grt.tlc**[1]	**ert.tlc**[1]	**ert_shrlib.tlc**[1]	**rtwsfcn.tlc**[1]	**rsim.tlc**[1]	**sldrt.tlc**[1]	**slrt.tlc**[1]	**Other**[1]
(noninlined)								
S-function (inlined)	X	X		X	X	X	X	X
Minimize RAM/ROM usage		X					X[2]	X
Supports external mode	X	X			X	X	X	
Rapid prototyping	X					X	X	X
Production code		X					X[2]	X[3]
Batch parameter tuning and Monte Carlo methods			X		X			
System-level Simulator			X					
Executes in hard real time	X[3]	X[3]				X	X	X[5]
Non-real-time executable included	X	X			X			
Multiple instances of model	X[4, 5]	X[4, 5]		X[4]			X[4, 5]	X[4, 5]

	System Target Files (STF)							
Feature	**grt.tlc**[1]	**ert.tlc**[1]	**ert_shrlib.tlc**[1]	**rtwsfcn.tlc**[1]	**rsim.tlc**[1]	**sldrt.tlc**[1]	**slrt.tlc**[1]	**Other**[1]
Supports variable-step solvers				X	X			
Supports SIL/PIL		X						X

[1] System Target Files: **grt.tlc** - generic real-time target, **ert.tlc** - embedded real-time target, **ert_shrlib.tlc** - embedded real-time target shared library), **rtwsfcn.tlc** - S-Function, **rsim.tlc** - rapid simulation, **sldrt.tlc** - Simulink Desktop Real-Time, **slrt.tlc** - Simulink Real-Time, and **Other** - The embedded real-time capabilities in Simulink Coder support other system target files.

[2]Does not apply to GRT-based system target files. Applies only to an ERT-based system target files.

[3]The default GRT and ERT `rt_main` files emulate execution of hard real time, and when explicitly connected to a real-time clock execute in hard real time.

[4]You can generate code for multiple instances of a Stateflow chart or subsystem containing a chart, except when the chart contains exported graphical functions or the Stateflow model contains machine parented events.

[5]You must select the value **Reusable function** for **Code interface packaging** (Simulink Coder) in the **Code Generation > Interface** pane of the Configuration Parameters dialog box.

More About

- “Select a System Target File” on page 30-2
- “Configure STF-Related Code Generation Parameters” on page 30-7
- “Configure a Code Replacement Library” on page 30-17
- “Configure Standard Math Library for Target System” on page 30-18

Internationalization Support in Simulink Coder

Internationalization and Code Generation

Internationalization support in software development tooling is vital for enabling efficient globalization. If there is any possibility of future collaboration with others across locales, consider internationalization from project inception. Internationalization can prevent rework or having to develop a new model design. The relevant requirement concerns locale settings.

In this section...

Locale Settings

On a computer, a locale setting defines the language (character set encoding) for the user interface and the display formats for information such as time, date, and currency. The encoding dictates the number of characters that a locale can render. For example, the US-ASCII coded character set (codeset) defines 128 characters. A Unicode® codeset, such as UTF-8, defines more than 1,100,000 characters.

For code generation, the locale setting determines the character set encoding of generated file content. To avoid garbled text or incorrectly displayed characters, the locale setting for your MATLAB session must be compatible with the setting for your compiler and operating system. For information on finding and changing the operating system setting, see "Internationalization" (MATLAB) or see the operating system documentation.

To check a model for characters that cannot be represented in the locale setting of your current MATLAB session, use the Simulink Model Advisor check "Check model for foreign characters" (Simulink).

Prepare to Generate Code for Mixed Languages and Locales

To prepare to generate code for a model, identify:

- The operating system locale.
- The locale of the MATLAB session.
- Code generation requirements for:
  - Target Language Compiler files
  - Code generation template files that include comments (requires Embedded Coder)

Character Set Limitations

Target language compiler files support user default encoding only. To produce international, custom generated code that is portable, use the 7-bit ASCII character set.

XML Escape Sequence Replacements

The code generator replaces characters that are not represented in the character set encoding of a model with XML escape sequences. Escape sequence replacements occur for block, signal, and Stateflow object names that appear in:

- Generated code comments
- Code generation reports
- Block paths logged to MAT-files
- Block paths logged to C API files *`model`*`_capi.c` (or `.cpp`) and *`model`*`_capi.h`

Generate and Review Code with Mixed Languages and Mixed Locales

This example shows how to use the code generator to generate and review code for use in mixed languages and mixed locales.

Before using this example, see “Internationalization and Code Generation” (Simulink Coder) or “Internationalization and Code Generation”.

The `rtwdemo_unicode` model configuration uses the Embedded Coder (R) `ert.tlc` system target file. To see internationalization and localization support with Simulink Coder®, configure the model to use the `grt.tlc` system target file. The example indicates the support that is specific to Embedded Coder® (for example, code generation templates).

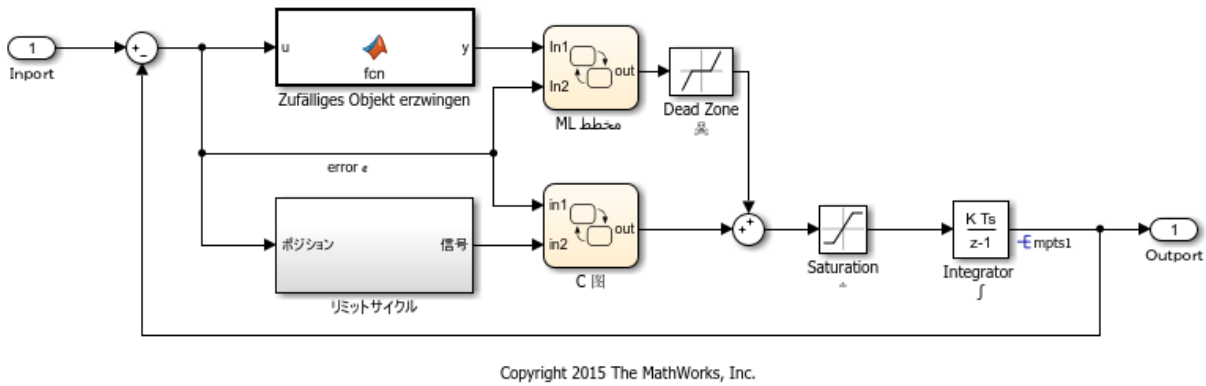
The model configuration specifies files and settings that control how the code generator handles localization for:

- C and C++ API interfaces
- Code generation template (CGT) files (requires Embedded Coder®)
- Target Language Compiler (TLC) files that apply code customizations (requires Embedded Coder®)

Open the example model `rtwdemo_unicode`.

Labels in the model appear in multiple languages (Arabic, Chinese, English, German, and Japanese) and various Unicode symbols.

```
model = 'rtwdemo_unicode';
open_system(model);
```



Verify Locale Settings

Verify that the locale setting for your MATLAB® software is compatible with your compiler. See the documentation for your operating system or the following MATLAB documentation:

- “Set Locale on Windows Platforms” (MATLAB)
- “Set Locale on Linux Platforms” (MATLAB)
- “Set Locale on Mac Platforms” (MATLAB)

Verify Model for Use of Foreign Characters

to verify the model for characters that the code generator cannot represent in the model's current character set encoding, use the Simulink® Model Advisor check **Check model for foreign characters**.

1. Open the Model Advisor in Simulink®. Select **Analysis** > **Model Advisor** > **Model Advisor**. Or, in the Command Window, type:

```
modeladvisor('rtwdemo_unicode')
```

```
Loading Model Advisor cache…
Warning: Cannot load an object of class
'SDRSLHDLWAPlugin':
Its class cannot be found.
Warning: Cannot load an object of class
'SDRSLHDLWAPlugin':
Its class cannot be found.
Model Advisor cache loaded. For new customizations, to update the cache, use the Adviso
Updating Model Advisor cache…
Model Advisor cache updated. For new customizations, to update the cache, use the Advis
```

2. Expand **By Product**.

3. Expand **Simulink**.

4. Select **Check model for foreign characters**

5. Click **Run This Check**.

6. Review the results. Several warnings appear. Verify that the characters in the model can be represented in the current character set encoding.

7. Close the Model Advisor.

Code Generation Template Files

To use a code generation template file with unicode characters when generating code, complete these steps (requires Embedded Coder®). Otherwise, go to the next section.

1. Open the Configuration Parameters dialog box.

2. Navigate to the **Code Generation > Template** pane. The model is configured to use the code generation template file `rtwdemo_unicode.cgt`. That file adds comments to the top of generated code files. For the code generator to apply escape sequence replacements for the `.cgt` file, enable replacements by specifying:

```
<encodingIn = "encoding-name">
```

3. Open the file `/toolbox/rtw/rtwdemos/rtwdemo_unicode.cgt`.

```
edit rtwdemo_unicode.cgt
```

4. Find the line of code that enables escape sequence replacements for the character set encoding `UTF-8`.

```
<encodingIn = "UTF-8">
```

5. Close the file `/toolbox/rtw/rtwdemos/rtwdemo_unicode.cgt`.

Generated File Customization Template

To use file customization templates with unicode characters when generating code, complete these steps (requires Embedded Coder®). Otherwise, go to the next section.

You can specify customizations to generated code files by using TLC code. TLC files support user default encoding only. To produce international custom generated code that is portable, use the 7-bit ASCII character set.

1. Open the Configuration Parameters dialog box.

2. Navigate to the **Code Generation > Template** pane. The model is configured to use the code customization file `example_file_process.tlc`. That file customizes the generated code just before the code generator writes the code files. For example, the file adds a C source file, corresponding include file, and `#define` and `#include` statements.

3. Open the file `/toolbox/rtw/rtwdemos/example_file_process.tlc`.

```
edit example_file_process.tlc
```

4. Before generating code, uncomment the following line of code:

```
%% %assign ERTCustomFileTest = TLC_TRUE%
```

5. Close the file `/toolbox/rtw/rtwdemos/rtwdemo_unicode.cgt`.

Generate C Code

Generate C code and a code generation report.

```
evalc('rtwbuild(''rtwdemo_unicode'')');
```

Review the Generated Code

For characters that are not in the current MATLAB® character set encoding, the code generator uses escape sequence replacements to render characters correctly in the code generation report.

1. If the code generation report for model `rtwdemo_unicode` is not open, in the Command Window, type:

```
coder.report.open('rtwdemo_unicode')
```

2. Review the generated code in `rtwdemo_unicode.c` and `rtwdemo_unicode.h`. Names of model elements appear in code comments as replacement names in the local language.

3. Open the Traceability Report. The report maintains traceability information, even when the name contains characters that are not represented in the current encoding. Names of model elements appear in the report as replacement names in the local language.

4. Scroll down to and click the code location link for the first Chart (`State 'Selection' <S2>:23`). The report view changes to show the corresponding code in `rtwdemo_unicode.c`.

5. In the code comment, click the `<S2>:23` link. The model window shows the chart in a new tab.

6. In the model window, right-click that chart. Select **C/C++ Code > Navigate to C/C++ Code**. The report view changes to show the named constant section of code for that chart.

7. Close the code generation report, Model Advisor, and model. In the Command Window, type:

```
coder.report.close();
bdclose('all');
```

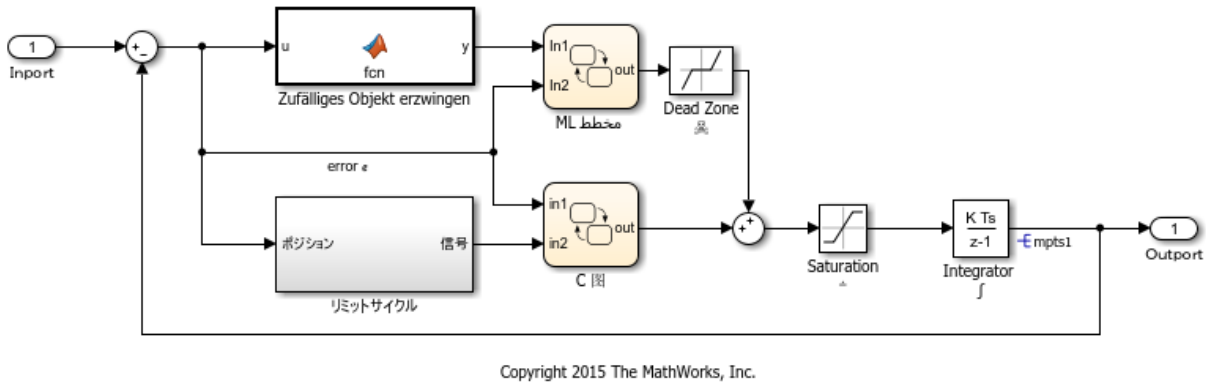
Generate C++ Code

Generate C++ code and a code generation report.

1. Open the model.

```
model = 'rtwdemo_unicode';
```

```
open_system(model);
```



2. Change **Configuration Parameters** > **Code Generation** > **Language** to `C++`. Or, in the Command Window, type:

```
set_param('rtwdemo_unicode','TargetLang','C++');
```

3. Change **Configuration Parameters** > **Code Generation** > **Interface** > **Code interface packaging** to `C++ class`. Or, in the Command Window, type:

```
set_param('rtwdemo_unicode','CodeInterfacePackaging','C++ class');
```

4. Generate C++ code and a code generation report.

```
evalc('rtwrebuild(''rtwdemo_unicode'')');
```

5. To see internationalization and localization support, review the generated code. See **Review the Generated Code**.

6. Close the code generation report and model. In the Command Window, type:

```
coder.report.close();
bdclose('all');
```

More About

- "Locale Settings for MATLAB Process" (MATLAB)

32

Internationalization Support in Embedded Coder

Internationalization and Code Generation

Internationalization support in software development tooling is vital to enabling efficient globalization. If there is a remote possibility that you could collaborate in the future with others across locales, consider internationalization from project inception. Internationalization can prevent rework or having to develop a new model design. The relevant requirement concerns locale settings.

In this section...

Locale Settings

On a computer, a locale setting defines the language (character set encoding) for the user interface and the display formats for information such as time, date, and currency. The encoding dictates the number of characters that a locale can render. For example, the US-ASCII coded character set (codeset) defines 128 characters. A Unicode codeset, such as UTF-8, defines more than 1,100,000 characters.

For code generation, the locale setting determines the character set encoding of generated file content. To avoid garbled text or incorrectly displayed characters, the locale setting for your MATLAB session must be compatible with the setting for your compiler and operating system. For information on finding and changing the operating system setting, see "Internationalization" (MATLAB) or see the operating system documentation.

To check a model for characters that cannot be represented in the locale setting of your current MATLAB session, use the Simulink Model Advisor check "Check model for foreign characters" (Simulink).

Prepare to Generate Code for Mixed Languages and Locales

To prepare to generate code for a model, identify:

- The operating system locale.
- The locale of the MATLAB session.
- Code generation requirements for:
  - Target Language Compiler files
  - Code generation template files that include comments (requires Embedded Coder)

Character Set Limitations

Target language compiler files support user default encoding only. To produce international, custom generated code that is portable, use the 7-bit ASCII character set.

XML Escape Sequence Replacements

The code generator replaces characters that are not represented in the character set encoding of a model with XML escape sequences. Escape sequence replacements occur for block, signal, and Stateflow object names that appear in:

- Generated code comments
- Code generation reports
- Block paths logged to MAT-files
- Block paths logged to C API files *`model`*`_capi.c` (or `.cpp`) and *`model`*`_capi.h`

CGT Files and XML Escape Sequence Replacements

The code generator replaces characters that are not represented in the character set encoding for a model with XML escape sequences. Escape sequence replacements occur for block, signal, and Stateflow object names that appear in Comments in code generation template (CGT) files.

By default, code generation template files do not contain character set encoding information. The operating system reads the files, using its current encoding, regardless of the encoding that you use to write the file. You can enable escape sequence replacements by adding the following token at the top of the template file:

```
<encodingIn = "encoding">
```

Replace *encoding* with a string that names a standard character encoding scheme, such as `UTF-8`, `ISO-8859–1`, or `windows-1251`.

This example shows content from the file rtwdemo_unicode.cgt in a MATLAB session for `windows-1251`. The example uses the `encodingIn` token to set the encoding to `UTF–8`, which is the correct value for code generation.

```
17  <encodingIn = "UTF-8">
18  <FileBanner style = "box">
19  File: %<FileName>
20
21  Code generated for Simulink model '%<ModelName>'.
22
23  Model version                  : %<ModelVersion>
24  Simulink Coder version         : %<RTWFileVersion>
25  TLC version                    : %<TLCVersion>
26  C/C++ source code generated on : %<SourceGeneratedOn>
27
28  You can customize this banner by specifying a different template.
29   Unicode characters written in cgt file:
30   ï¼³ï¼●ï¼‰ï¼": â™  â™¡ â™¢ â™£ â™¤ â™¥ â™¦
31   Greek Symbols: â™ƒ â™„ â™… â™† â™‡ â™ˆ â™‰ â™Š â™‹ â™Œ â™ â™Ž â™ â™ â™‘ â™’ â™“
```

Generate and Review Code with Mixed Languages and Mixed Locales

This example shows how to use the code generator to generate and review code for use in mixed languages and mixed locales.

Before using this example, see “Internationalization and Code Generation” (Simulink Coder) or “Internationalization and Code Generation”.

The `rtwdemo_unicode` model configuration uses the Embedded Coder (R) `ert.tlc` system target file. To see internationalization and localization support with Simulink Coder®, configure the model to use the `grt.tlc` system target file. The example indicates the support that is specific to Embedded Coder® (for example, code generation templates).

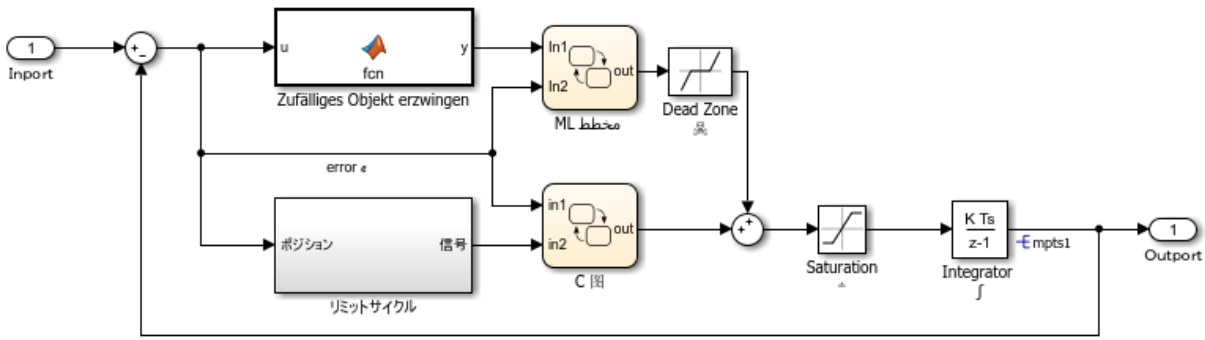
The model configuration specifies files and settings that control how the code generator handles localization for:

- C and C++ API interfaces
- Code generation template (CGT) files (requires Embedded Coder®)
- Target Language Compiler (TLC) files that apply code customizations (requires Embedded Coder®)

Open the example model `rtwdemo_unicode`.

Labels in the model appear in multiple languages (Arabic, Chinese, English, German, and Japanese) and various Unicode symbols.

```
model = 'rtwdemo_unicode';
open_system(model);
```



Verify Locale Settings

Verify that the locale setting for your MATLAB® software is compatible with your compiler. See the documentation for your operating system or the following MATLAB documentation:

- “Set Locale on Windows Platforms” (MATLAB)
- “Set Locale on Linux Platforms” (MATLAB)
- “Set Locale on Mac Platforms” (MATLAB)

Verify Model for Use of Foreign Characters

to verify the model for characters that the code generator cannot represent in the model's current character set encoding, use the Simulink® Model Advisor check **Check model for foreign characters**.

1. Open the Model Advisor in Simulink®. Select **Analysis** > **Model Advisor** > **Model Advisor**. Or, in the Command Window, type:

```
modeladvisor('rtwdemo_unicode')
```

```
Loading Model Advisor cache…
Warning: Cannot load an object of class
'SDRSLHDLWAPlugin':
Its class cannot be found.
Warning: Cannot load an object of class
'SDRSLHDLWAPlugin':
Its class cannot be found.
Model Advisor cache loaded. For new customizations, to update the cache, use the Adviso
Updating Model Advisor cache…
Model Advisor cache updated. For new customizations, to update the cache, use the Advis
```

2. Expand **By Product**.

3. Expand **Simulink**.

4. Select **Check model for foreign characters**

5. Click **Run This Check**.

6. Review the results. Several warnings appear. Verify that the characters in the model can be represented in the current character set encoding.

7. Close the Model Advisor.

Code Generation Template Files

To use a code generation template file with unicode characters when generating code, complete these steps (requires Embedded Coder®). Otherwise, go to the next section.

1. Open the Configuration Parameters dialog box.

2. Navigate to the **Code Generation > Template** pane. The model is configured to use the code generation template file `rtwdemo_unicode.cgt`. That file adds comments to the top of generated code files. For the code generator to apply escape sequence replacements for the `.cgt` file, enable replacements by specifying:

```
<encodingIn = "encoding-name">
```

3. Open the file `/toolbox/rtw/rtwdemos/rtwdemo_unicode.cgt`.

```
edit rtwdemo_unicode.cgt
```

4. Find the line of code that enables escape sequence replacements for the character set encoding `UTF-8`.

```
<encodingIn = "UTF-8">
```

5. Close the file `/toolbox/rtw/rtwdemos/rtwdemo_unicode.cgt`.

Generated File Customization Template

To use file customization templates with unicode characters when generating code, complete these steps (requires Embedded Coder®). Otherwise, go to the next section.

You can specify customizations to generated code files by using TLC code. TLC files support user default encoding only. To produce international custom generated code that is portable, use the 7-bit ASCII character set.

1. Open the Configuration Parameters dialog box.

2. Navigate to the **Code Generation > Template** pane. The model is configured to use the code customization file `example_file_process.tlc`. That file customizes the generated code just before the code generator writes the code files. For example, the file adds a C source file, corresponding include file, and `#define` and `#include` statements.

3. Open the file `/toolbox/rtw/rtwdemos/example_file_process.tlc`.

```
edit example_file_process.tlc
```

4. Before generating code, uncomment the following line of code:

```
%% %assign ERTCustomFileTest = TLC_TRUE%
```

5. Close the file `/toolbox/rtw/rtwdemos/rtwdemo_unicode.cgt`.

Generate C Code

Generate C code and a code generation report.

```
evalc('rtwbuild(''rtwdemo_unicode'')');
```

Review the Generated Code

For characters that are not in the current MATLAB® character set encoding, the code generator uses escape sequence replacements to render characters correctly in the code generation report.

1. If the code generation report for model `rtwdemo_unicode` is not open, in the Command Window, type:

```
coder.report.open('rtwdemo_unicode')
```

2. Review the generated code in `rtwdemo_unicode.c` and `rtwdemo_unicode.h`. Names of model elements appear in code comments as replacement names in the local language.

3. Open the Traceability Report. The report maintains traceability information, even when the name contains characters that are not represented in the current encoding. Names of model elements appear in the report as replacement names in the local language.

4. Scroll down to and click the code location link for the first Chart (`State 'Selection' <S2>:23`). The report view changes to show the corresponding code in `rtwdemo_unicode.c`.

5. In the code comment, click the `<S2>:23` link. The model window shows the chart in a new tab.

6. In the model window, right-click that chart. Select **C/C++ Code > Navigate to C/C++ Code**. The report view changes to show the named constant section of code for that chart.

7. Close the code generation report, Model Advisor, and model. In the Command Window, type:

```
coder.report.close();
bdclose('all');
```

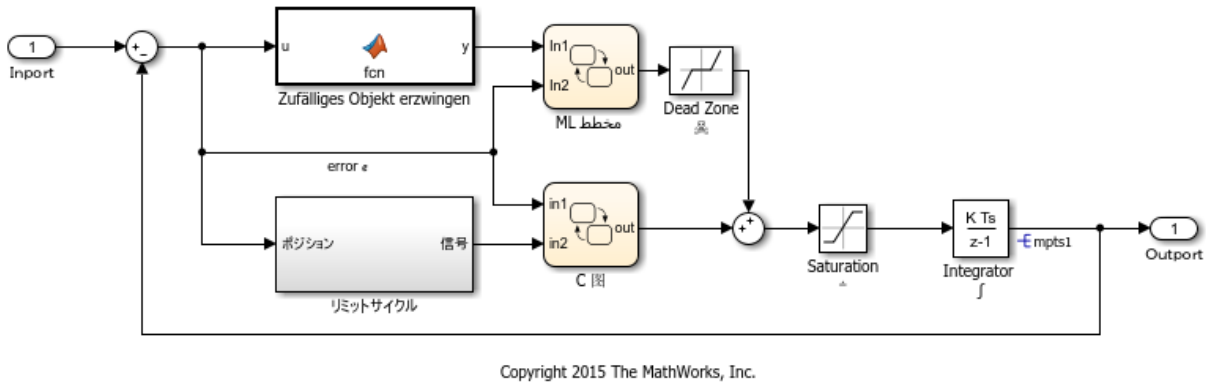
Generate C++ Code

Generate C++ code and a code generation report.

1. Open the model.

```
model = 'rtwdemo_unicode';
```

```
open_system(model);
```



2. Change **Configuration Parameters** > **Code Generation** > **Language** to `C++`. Or, in the Command Window, type:

```
set_param('rtwdemo_unicode','TargetLang','C++');
```

3. Change **Configuration Parameters** > **Code Generation** > **Interface** > **Code interface packaging** to `C++ class`. Or, in the Command Window, type:

```
set_param('rtwdemo_unicode','CodeInterfacePackaging','C++ class');
```

4. Generate C++ code and a code generation report.

```
evalc('rtwrebuild(''rtwdemo_unicode'')');
```

5. To see internationalization and localization support, review the generated code. See **Review the Generated Code**.

6. Close the code generation report and model. In the Command Window, type:

```
coder.report.close();
bdclose('all');
```

More About

- "Locale Settings for MATLAB Process" (MATLAB)

Source Code Generation in Simulink Coder

Configure Model, Generate Code, and Simulate

In this section...

About This Example

Learning Objectives

- Learn about the functional behavior of the example model.
- Learn about the role of the example test harness and its components.
- Run simulation tests on a model.

Prerequisites

- Ability to open and modify Simulink models and subsystems.
- Understand subsystems and how to view subsystem details.
- Understand referenced models and how to view referenced model details.
- Ability to set model configuration parameters.

Required Files

Before you use each example model file, place a copy in a writable location and add it to your MATLAB path.

- `rtwdemo_throttlecntrl` model file
- `rtwdemo_throttlecntrl_testharness` model file

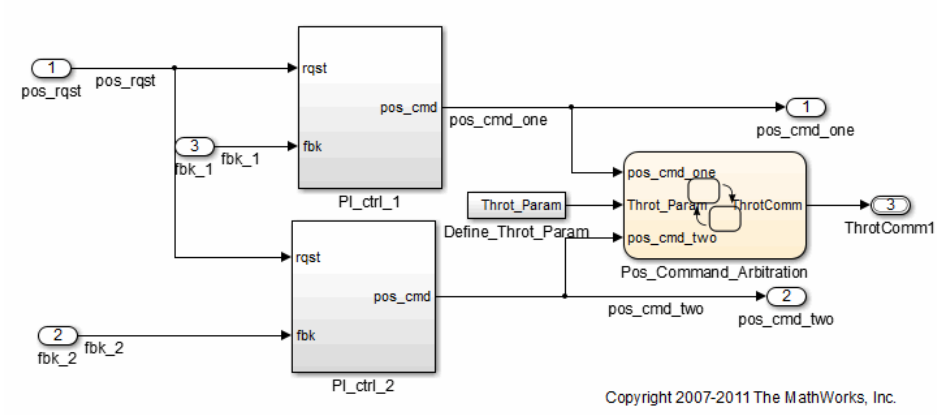
Functional Design of the Model

This example uses a simple, but functionally complete, example model of a throttle controller. The model features redundant control algorithms. The model highlights a standard model structure and a set of basic blocks in algorithm design.

View the Top Model

Open `rtwdemo_throttlecntrl` and save a copy as `throttlecntrl` in a writable location on your MATLAB path.

Note: This model uses Stateflow software.



The top level of the model consists of the following elements:

Subsystems	`PI_ctrl_1`
	`PI_ctrl_2`
	`Define_Throt_Param`
	`Pos_Command_Arbitration`

Top-level input	`pos_rqst`
	`fbk_1`
	`fbk_2`

Top-level output	`pos_cmd_one`
	`pos_cmd_two`
	`ThrotComm1`

Signal routing	
Omit blocks that change the value of a signal, such as Sum and Integrator	

The layout uses a basic architectural style for models:

- Separation of calculations from signal routing (lines and buses)
- Partitioning into subsystems

You can apply this style to a wide range of models.

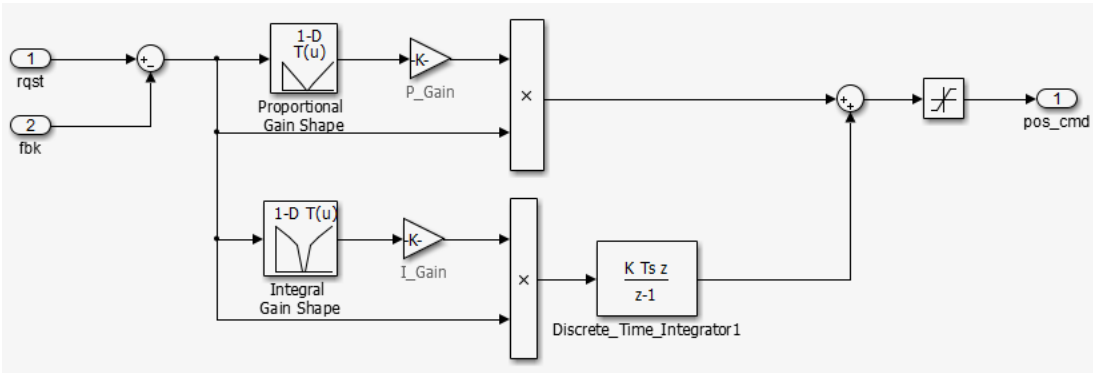
View the Subsystems

Explore two of the subsystems in the top model.

1. If not already open, open `throttlecntrl`.

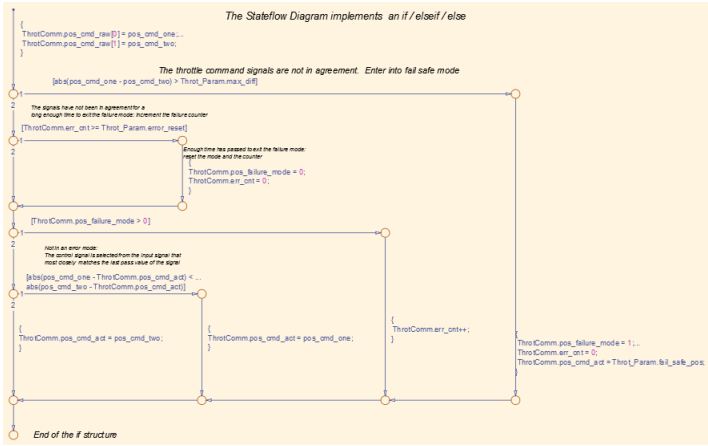
   Two subsystems in the top model represent proportional-integral (PI) controllers, `PI_ctrl_1` and `PI_ctrl_2`. At this stage, these identical subsystems, use identical data. If you have Embedded Coder, you can use these subsystems in an example that shows how to "Customize Function Interface and File Packaging".

2. Open the `PI_ctrl_1` subsystem.



The PI controllers in the model are from a *library*, a group of related blocks or models for reuse. Libraries provide one of two methods for including and reusing models. The second method, model referencing, is described in "Simulation Test Environment" on page 33-5. You cannot edit a block that you add to a model from a library. Edit the block in the library so that instances of the block in different models remain consistent.

3 Open the `Pos_Command_Arbitration` subsystem. This Stateflow chart performs basic error checking on the two command signals. If the command signals are too far apart, the Stateflow diagram sets the output to a `fail_safe` position.



4 Close `throttlecntrl`.

Simulation Test Environment

To test the throttle controller algorithm, incorporate it into a *test harness*. A test harness is a model that evaluates the control algorithm and offers the following benefits:

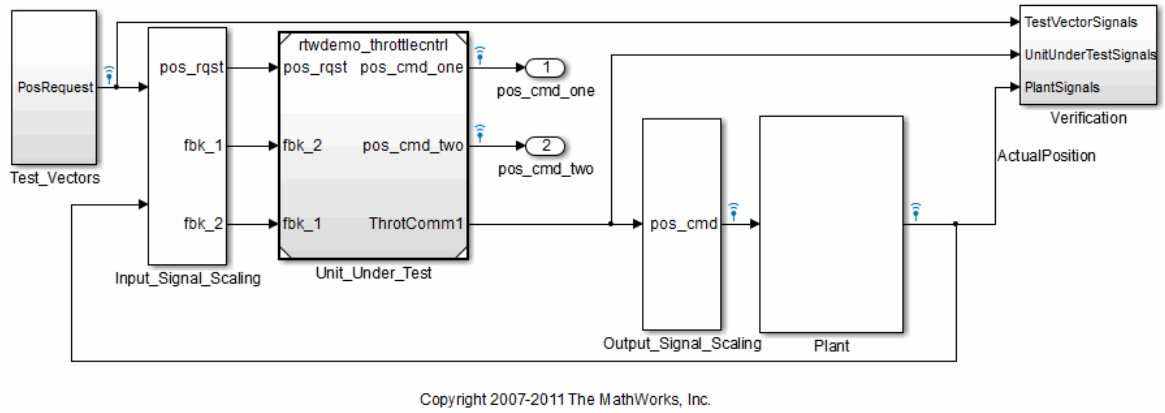
- Separates test data from the control algorithm.
- Separates the plant or feedback model from the control algorithm.
- Provides a reusable environment for multiple versions of the control algorithm.

The test harness model for this example implements a common simulation testing environment consisting of the following parts:

- Unit under test
- Test vector source
- Evaluation and logging
- Plant or feedback system
- Input and output scaling

Explore the simulation testing environment.

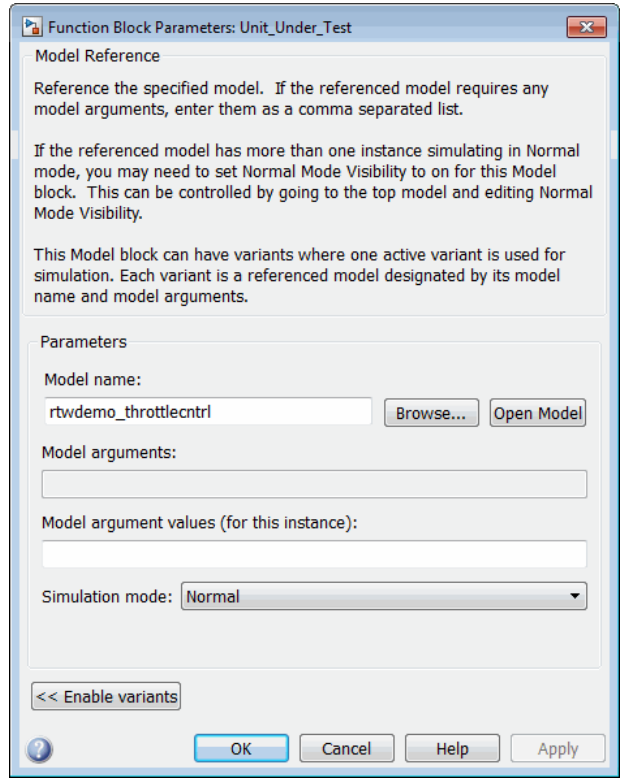
1 Open the test harness model `rtwdemo_throttlecntrl_testharness` and save a copy as `throttlecntrl_testharness` in a writable location on your MATLAB path.



2 Set up your `throttlecntrl` model as the control algorithm of the test harness.

a Open the `Unit_Under_Test` block and view the control algorithm.

b View the model reference parameters by right-clicking the `Unit_Under_Test` block and selecting **Block Parameters (ModelReference)**.



`rtwdemo_throttlecntrl` appears as the name of the referenced model.

c Change the value of **Model name** to `throttlecntrl`.

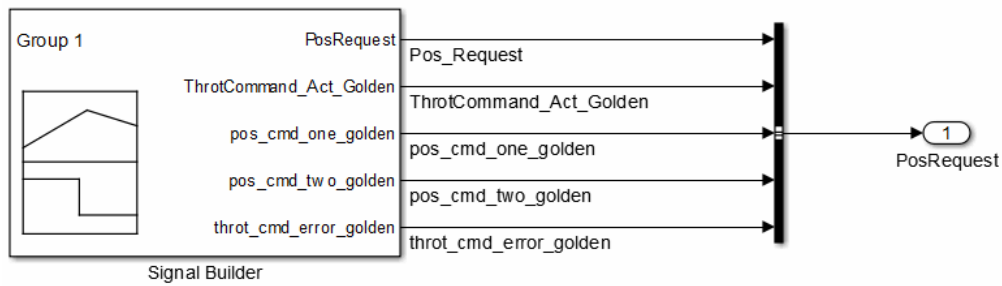
d Update the test harness model diagram by clicking **Simulation** > **Update Diagram**.

The control algorithm is the *unit under test*, as indicated by the name of the Model block, `Unit_Under_Test`.

The Model block provides a method for reusing components. From the top model, it allows you to reference other models (directly or indirectly) as *compiled functions*. By default, Simulink software recompiles the model when the referenced models change. Compiled functions have the following advantages over libraries:

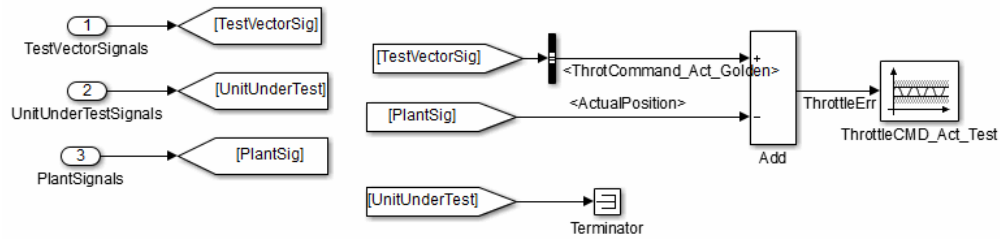
- Simulation time is faster for large models.
- You can directly simulate compiled functions.
- Simulation requires less memory. Only one copy of the compiled model is in memory, even when the model is referenced multiple times.

3 Open the *test vector source*, implemented in this test harness as the `Test_Vectors` subsystem.



The subsystem uses a Signal Builder block for the test vector source. The block has data that drives the simulation (`PosRequest`) and provides the expected results used by the `Verification` subsystem. This example test harness uses only one set of test data. Typically, create a test suite that fully exercises the system.

4 Open the *evaluation and logging* subsystem, implemented in this test harness as subsystem `Verification`.

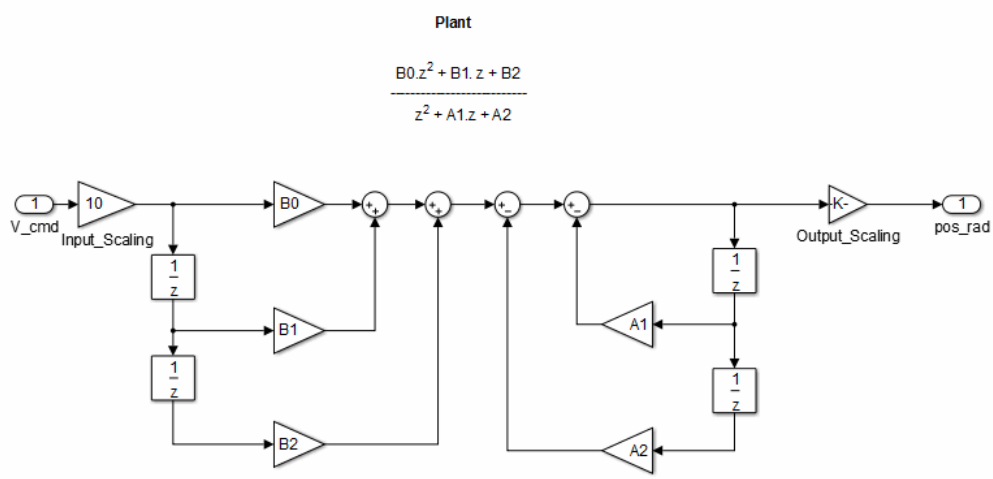


A test harness compares control algorithm simulation results against *golden data* — test results that exhibit the desired behavior for the control algorithm as certified by an expert. In the `Verification` subsystem, an Assertion block compares the simulated throttle value position from the plant against the golden value from the

test harness. If the difference between the two signals is greater than 5%, the test fails and the Assertion block stops the simulation.

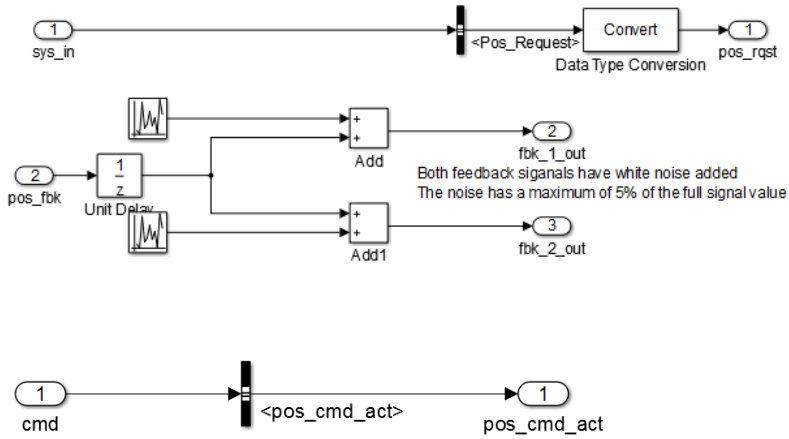
Alternatively, you can evaluate the simulation data after the simulation completes execution. Perform the evaluation with either MATLAB scripts or third-party tools. Post-execution evaluation provides greater flexibility in the analysis of data. However, it requires waiting until execution is complete. Combining the two methods can provide a highly flexible and efficient test environment.

5 Open the *plant or feedback system*, implemented in this test harness as the `Plant` subsystem.



The `Plant` subsystem models the throttle dynamics with a transfer function in canonical form. You can create plant models to varying levels of fidelity. It is common to use different plant models at different stages of testing.

6 Open the *input and output scaling* subsystems, implemented in this test harness as `Input_Signal_Scaling` and `Output_Signal_Scaling`.



The subsystems that scale input and output perform the following primary functions:

- Select input signals to route to the unit under test.
- Select output signals to route to the plant.
- Rescale signals between engineering units and units that are writable for the unit under test.
- Handle rate transitions between the plant and the unit under test.

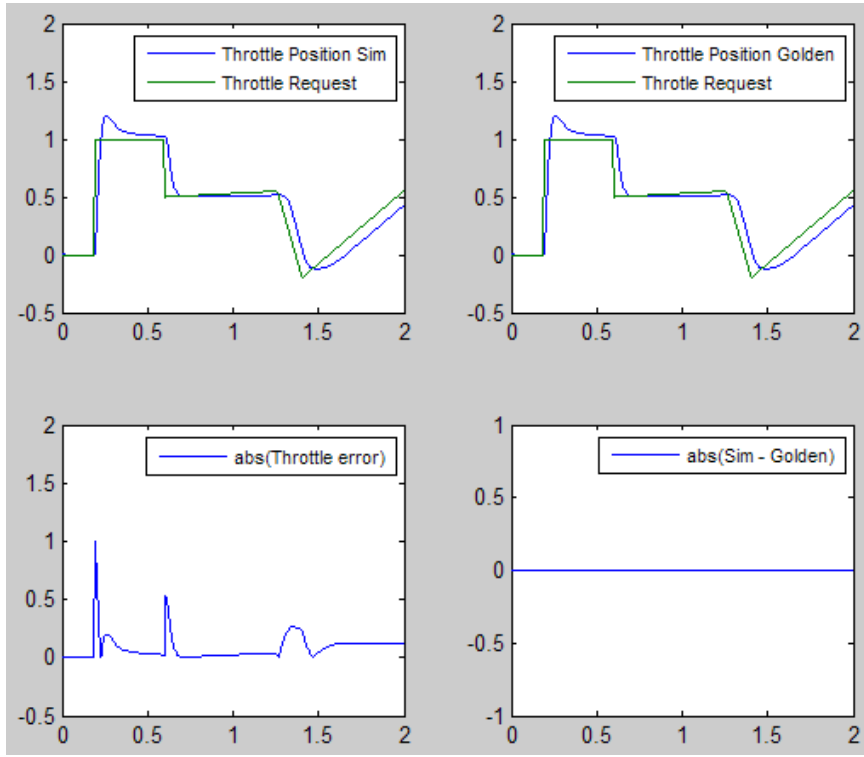
7 Save and close `throttlecntrl_testharness`.

Run Simulation Tests

1 Check that your working folder is set to a writable folder, such as the folder into which you placed copies of the example model files.

2 Open your copy of the test harness model, `throttlecntrl_testharness`.

3 Start a test harness model simulation. When the simulation is complete, the following results appear.



The lower-right hand plot shows the difference between the expected (golden) throttle position and the throttle position that the plant calculates. If the difference between the two values is greater than ±0.05, the simulation stops.

4 Save and close throttle controller and test harness models.

Key Points

- A basic model architecture separates calculations from signal routing and partitions the model into subsystems
- Two options for model reuse include block libraries and model referencing.
- If you represent your control algorithm in a test harness as a Model block, specify the name of the control algorithm model in the Model Reference Parameters dialog box.

- A test harness is a model that evaluates a control algorithm. Typically, a harness consists of a unit under test, a test vector source, evaluation and logging, a plant or feedback system, and input and output scaling components.
- The unit under test is the control algorithm being tested.
- The test vector source provides the data that drives the simulation which generates results used for verification.
- During verification, the test harness compares control algorithm simulation results against golden data and logs the results.
- The plant or feedback component of a test harness models the environment that is being controlled.
- When developing a test harness,
  - Scale input and output components.
  - Select input signals to route to the unit under test.
  - Select output signals to route to the plant.
  - Rescale signals between engineering units and units that are writable for the unit under test.
  - Handle rate transitions between the plant and the unit under test.
- Before running simulation or completing verification, consider checking a model with the Model Advisor.

Learn More

- “Support Model Referencing” (Simulink Coder)
- “Code Generation” (Simulink Coder)
- “Signal Groups” (Simulink)

Configure Model and Generate Code

In this section...

About This Example

Learning Objectives

- Configure a model for code generation.
- Apply model checking tools to discover conditions and configuration settings resulting in generation of inaccurate or inefficient code.
- Generate code from a model.
- Locate and identify generated code files.
- Review generated code.

Prerequisites

- Ability to open and modify Simulink models and subsystems.
- Ability to set model configuration parameters.
- Ability to use the Simulink Model Advisor.
- Ability to read C code.
- An installed, supported C compiler.

Required Files

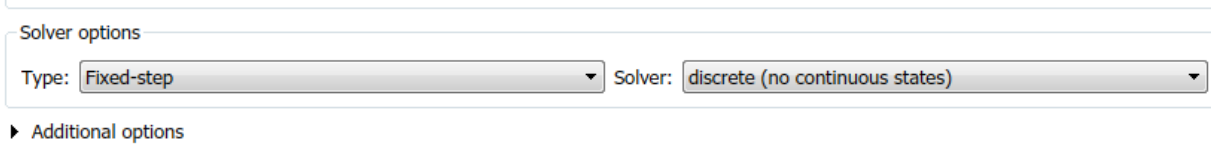
`rtwdemo_throttlecntrl` model file

Configure the Model for Code Generation

Model configuration parameters determine the method for generating the code and the resulting format.

1. Open `rtwdemo_throttlecntrl` and save a copy as `throttlecntrl` in a writable location on your MATLAB path.
2. Open the Configuration Parameters dialog box, **Solver** pane. To generate code for a model, you must configure the model to use a fixed-step solver. The following table shows the solver configuration for this example.

Parameter	Setting	Effect on Generated Code
Type	`Fixed-step`	Maintains a constant (fixed) step size, which is required for code generation
Solver	`discrete (no continuous states)`	Applies a fixed-step integration technique for computing the state derivative of the model
Fixed-step size	`.001`	Sets the base rate; must be the lowest common multiple of the rates in the system



3. Open the **Code Generation** > **General** pane and note that the **System target file** is set to `grt.tlc`.

Note: The GRT (Generic Real-Time Target) configuration requires a fixed-step solver. However, the `rsim.tlc` system target file supports variable step code generation.

The system target file (STF) defines an environment for generating and building code for execution on a certain hardware or operating system platform. For example, one property of a system target file is the value for the `CodeFormat` TLC variable. The GRT configuration requires a fixed step solver and the rsim.tlc supports variable step code generation.

4 Open the **Code Generation** > **Custom Code** pane and under **Include list of additional**, select **Include directories**. The following path appears in the text field:

```
"$matlabroot$\toolbox\rtw\rtwdemos\EmbeddedCoderOverview\"
```

This folder includes files that are required to build an executable for the model.

5 Close the dialog box.

Save Your Model Configuration as a MATLAB Function

You can save the settings of model configuration parameters as a MATLAB function by using the `getActiveConfigSet` function. In the MATLAB Command Window, enter:

```
thcntrlAcs = getActiveConfigSet('throttlecntrl');
thcntrlAcs.saveAs('throttlecntrlModelConfig');
```

You can then use the resulting function (for example, `throttlecntrlModelConfig`) to:

- Archive the model configuration.
- Compare different model configurations by using differencing tools.
- Set the configuration of other models.

For example, you can set the configuration of model `myModel` to match the configuration of the throttle controller model by opening `myModel` and entering:

```
myModelAcs = throttlecntrlModelConfig;
attachConfigSet('myModel', myModelAcs, true);
setActiveConfigSet('myModel', myModelAcs.Name);
```

For more information, see “Save a Configuration Set” (Simulink) and “Load a Saved Configuration Set” (Simulink).

Check Model Conditions and Configuration Settings

Before generating code for a model, use the Simulink Model Advisor to check the model for conditions and configuration settings. This check finds issues that can result in inaccurate or inefficient code.

1. Open `throttlecntrl`.
2. Start the Model Advisor by selecting **Analysis** > **Model Advisor** > **Model Advisor**. A dialog box opens showing the model system hierarchy.
3. Click `throttlecntrl` and then click **OK**. The Model Advisor window opens.
4. Expand **By Product** and **Embedded Coder**. By default, checks that do not trigger an Update Diagram, with one exception, are selected.
5. In the left pane, select the remaining checks and select **Embedded Coder**.
6. In the right pane, select **Show report after run** and click **Run Selected Checks**. The report shows a **Run Summary** that flags check warnings.
7. Review the report. The warnings highlight issues for embedded systems. At this point, you can ignore them. For more information about reports, see "View Model Advisor Reports" (Simulink).

Generate Code for the Model

1. Open `throttlecntrl`.
2. In the Configuration Parameters dialog box, select **Code Generation** > **Generate code only** and click **Apply**.
3. On the **Code Generation** > **Report** pane, select **Create code generation report** and click **Apply**.
4. With the model open, initiate code generation and the build process for the model by using any of the following options:

   - Click the **Build Model** button.
   - Press **Ctrl+B**.
   - Select **Code** > **C/C++ Code** > **Build Model**.
   - Invoke the `rtwbuild` command from the MATLAB command line.
   - Invoke the `slbuild` command from the MATLAB command line.

Watch the messages that appear in the MATLAB Command Window. The code generator produces standard C and header files, and an HTML code generation report. The code generator places the files in a *build folder*, a subfolder named `throttlecntrl_grt_rtw` under your current working folder.

Review the Generated Code

1. Open Model Explorer, and in the **Model Hierarchy** pane, expand the node for the `throttlecntrl` model, and select the **Code for** node.
2. In the **Contents** pane, select **HTML Report**. Model Explorer displays the HTML code generation report for the throttle controller model.
3. In the HTML report, click the link for the generated C model file and review the generated code. Look for these items in the report:
   - Identification, version, timestamp, and configuration comments.
   - Links to help you navigate within and between files
   - Data definitions
   - Scheduler code
   - Controller code
   - Model initialization and termination functions
   - Call interface for the GRT system target file — output, update, initialization, start, and terminate
4. Save and close `throttlecntrl`.

Consider examining the following files. In the HTML report **Contents** pane, click the links. Or, in your working folder, explore the generated code subfolder.

File	Description
`throttlecntrl.c`	C file that contains the scheduler, controller, initialization, and interface code
`throttlecntrl_data.c`	C file that assigns values to generated data structures
`throttlecntrl.h`	Header file that defines data structures
`throttlecntrl_private.h`	Header file that defines data used only by the generated code

File	Description
`throttlecntrl_types.h`	Header file that defines the model data structure

For more information, see "Manage Build Process File Dependencies" (Simulink Coder).

At this point, consider logging data to a MAT-file. For an example, see "Log Data for Analysis" (Simulink Coder).

Generate an Executable

1. Open `throttlecntrl`.
2. In the Configuration Parameters dialog box, clear the **Code Generation** > **Generate code only** check box and click **Apply**.
3. Press **Ctrl+B**. Watch the messages in the MATLAB Command Window. The code generator uses a template make file associated with your system target file selection to create an executable file. You can run this program on your workstation, independent of external timing and events.
4. Check your working folder for the file`throttlecntrl.exe`.
5. Run the executable. In the Command Window, enter `!throttlecntrl`. The ! character passes the command that follows it to the operating system, which runs the standalone program.

   The program produces one line of output in the Command Window:

   ```
   ** starting the model **
   ```

   At this point, consider logging data to a MAT-file. For an example, see "Log Data for Analysis" (Simulink Coder).

Tip: For UNIX platforms, run the executable in the Command Window with the syntax `!./executable_name`. If preferred, run the executable from an OS shell with the syntax `./executable_name`. For more information, see "Run External Commands, Scripts, and Programs" (MATLAB).

Key Points

- To generate code, change the model configuration to specify a fixed-step solver then select a system target file. Using the `grt.tlc` file requires a fixed-step solver. If the model contains continuous time blocks, you can use a variable-step solver with the `rsim.tlc` system target file.
- After debugging a model, consider configuring a model with parameter inlining enabled.
- Use the `getActiveConfigSet` function to save a model configuration for future use or to apply it to another model.
- Before generating code, consider checking a model with the Model Advisor.
- The code generator places generated files in a subfolder (*model*`_grt_rtw`) of your working folder.

More About

- "Code Generation" (Simulink Coder)
- "Configuration Reuse" (Simulink)
- "Run Model Checks" (Simulink)

Configure Data Interface

About This Example

Learning Objectives

- Configure the data interface for code generated for a model.
- Control the name, data type, and data storage class of signals and parameters in generated code.

Prerequisites

- Understanding ways to represent and use data and signals in models.
- Familiarity with representing data constructs as data objects.
- Ability to read C code.

Required File

`rtwdemo_throttlecntrl_datainterface` model file

Declare Data

Most programming languages require that you *declare* data before using it. The declaration specifies the following information:

Data Attribute	Description
Scope	The region of the program that has access to the data
Duration	The period during which the data is resident in memory
Data type	The amount of memory allocated for the data
Initialization	An initial value, a pointer to memory, or NULL. If you do not provide an initial value, most compilers assign a zero value or a null pointer.

The following data types are supported for code generation.

Supported Data Types

Name	Description
`double`	Double-precision floating point
`single`	Single-precision floating point
`int8`	Signed 8-bit integer
`uint8`	Unsigned 8-bit integer
`int16`	Signed 16-bit integer
`uint16`	Unsigned 16-bit integer
`int32`	Signed 32-bit integer
`uint32`	Unsigned 32-bit integer
Fixed-point data types	8-, 16-, 32-bit word lengths

A *storage class* is the scope and duration of a data item. For more information about storage classes, see

- "Override Default Parameter Behavior by Creating Global Variables in the Generated Code" (Simulink Coder)
- "Signal Storage Class" (Simulink Coder)
- "Storage Classes for Block States" (Simulink Coder)

Use Data Objects

In Simulink models and Stateflow charts, the following methods are available for declaring data: *data objects* and *direct specification*. This example uses the data object method. Both methods allow full control over the data type and storage class. You can mix the two methods in a single model.

In the MATLAB and Simulink environment, you can use data objects in various ways. This example focuses on the following types of data objects:

- Signal
- Parameter
- Bus

To configure the data interface for your model using the data object method, in the MATLAB base workspace, you define data objects. Then, associate them with your

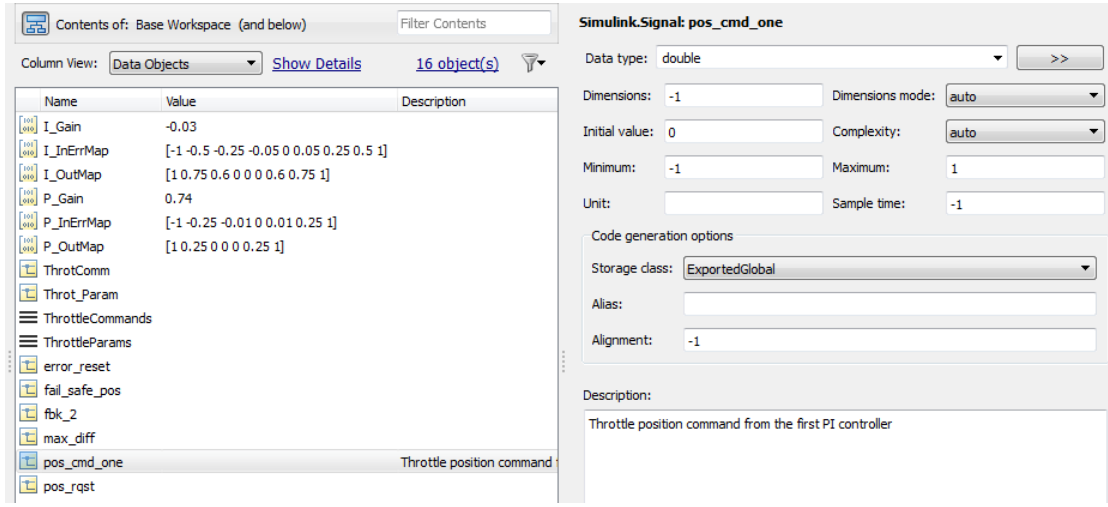
Simulink model or embedded Stateflow chart. When you build your model, the build process uses the associated base workspace data objects in the generated code.

You can set the values of the data object properties, which include:

- Data type
- Storage class
- Value (parameters)
- Initial value (signals)
- Alias (define a different name in the generated code)
- Dimension (typically inherited for parameters)
- Complexity (inherited for parameters)
- Unit (physical measurement unit)
- Minimum value
- Maximum value
- Description (used to document your data objects — does not affect simulation or code generation)

You can create and inspect base workspace data objects by entering commands in the MATLAB Command Window or by using Model Explorer. To explore base workspace signal data objects, use these steps:

1. Open `rtwdemo_throttlecntrl_datainterface` and save a copy as `throttlecntrl_datainterface` in a writable location on your MATLAB path.
2. Open Model Explorer.
3. Select **Base Workspace**.
4. Select the `pos_cmd_one` signal object for viewing.



You can also view the definition of a signal object. In the MATLAB Command Window, enter `pos_cmd_one`:

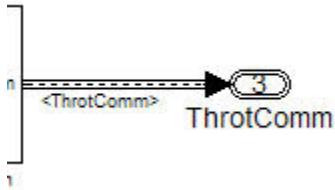
```
pos_cmd_one =

Signal with properties:

       CoderInfo: [1x1 Simulink.CoderInfo]
     Description: 'Throttle position command from the first PI controller'
        DataType: 'double'
             Min: -1
             Max: 1
            Unit: ''
      Dimensions: -1
  DimensionsMode: 'auto'
      Complexity: 'auto'
      SampleTime: -1
    InitialValue: '0'
```

5 To view other signal objects, in Model Explorer, click the object name or in the MATLAB Command Window, enter the object name. The following table summarizes object characteristics for some of the data objects in this model.

Object Characteristics	pos_cmd_one	pos_rqst	P_InErrMap	ThrotComm*	ThrottleCommands*
Description	Top-level output	Top-level input	Calibration parameter	Top-level output structure	Bus definition
Data type	Double	Double	Auto	Auto	Structure
Storage class	Exported global	Imported extern pointer	Constant	Exported global	None

* `ThrottleCommands` defines a Bus object; `ThrotComm` is an instantiation of the bus. If the bus is a nonvirtual bus, the signal generates a structure in the C code.

You can use a bus definition (`ThrottleCommands`) to instantiate multiple instances of the structure. In a model diagram, a bus object appears as a wide line with central dashes, as shown.



Add New Data Objects

You can create data objects for named signals, states, and parameters. To associate a data object with a construct, the construct must have a name.

To find constructs for which you can create data objects, use the Data Object Wizard. This tool finds the constructs and then creates the objects for you. The model includes two signals that are not associated with data objects: `fbk_1` and `pos_cmd_two`.

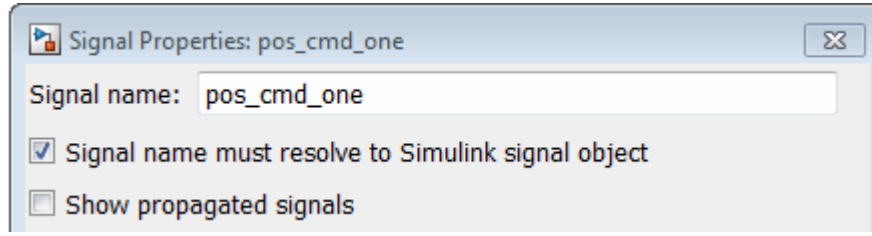
To find the signals and create data objects for them:

1 In the model window, select **Code > Data Objects > Data Object Wizard**. The Data Object Wizard dialog box opens.

2. To find candidate constructs, click **Find**. Constructs `fbk_1` and `pos_cmd_two` appear in the dialog box.
3. To select both constructs, click **Select All**.
4. In the table, under **Class**, make sure that each proposed data object uses the class `Simulink.Signal`. To change the class of the objects, click **Change Class**.
5. To create the data objects, click **Create**. Constructs `fbk_1` and `pos_cmd_two` are removed from the dialog box.
6. Close the Data Object Wizard.
7. In the **Contents** pane of the Model Explorer, find the newly created objects `fbk_1` and `pos_cmd_two`.

Enable Data Objects for Generated Code

1. Enable a signal to appear in generated code.
   a. In the model window, right-click the `pos_cmd_one` signal line and select **Properties**. A Signal Properties dialog box opens.
   b. Make sure that you select the **Signal name must resolve to Simulink signal object** parameter.



2. Enable signal object resolution for the signals in the model. In the MATLAB Command Window, enter:

```
disableimplicitsignalresolution('throttlecntrl_datainterface')
```

3. Save and close `throttlecntrl_datainterface`.

Effects of Simulation on Data Typing

In the throttle controller model, the data types are set to `double`. Because Simulink software uses the `double` data type for simulation, do not expect changes in the model

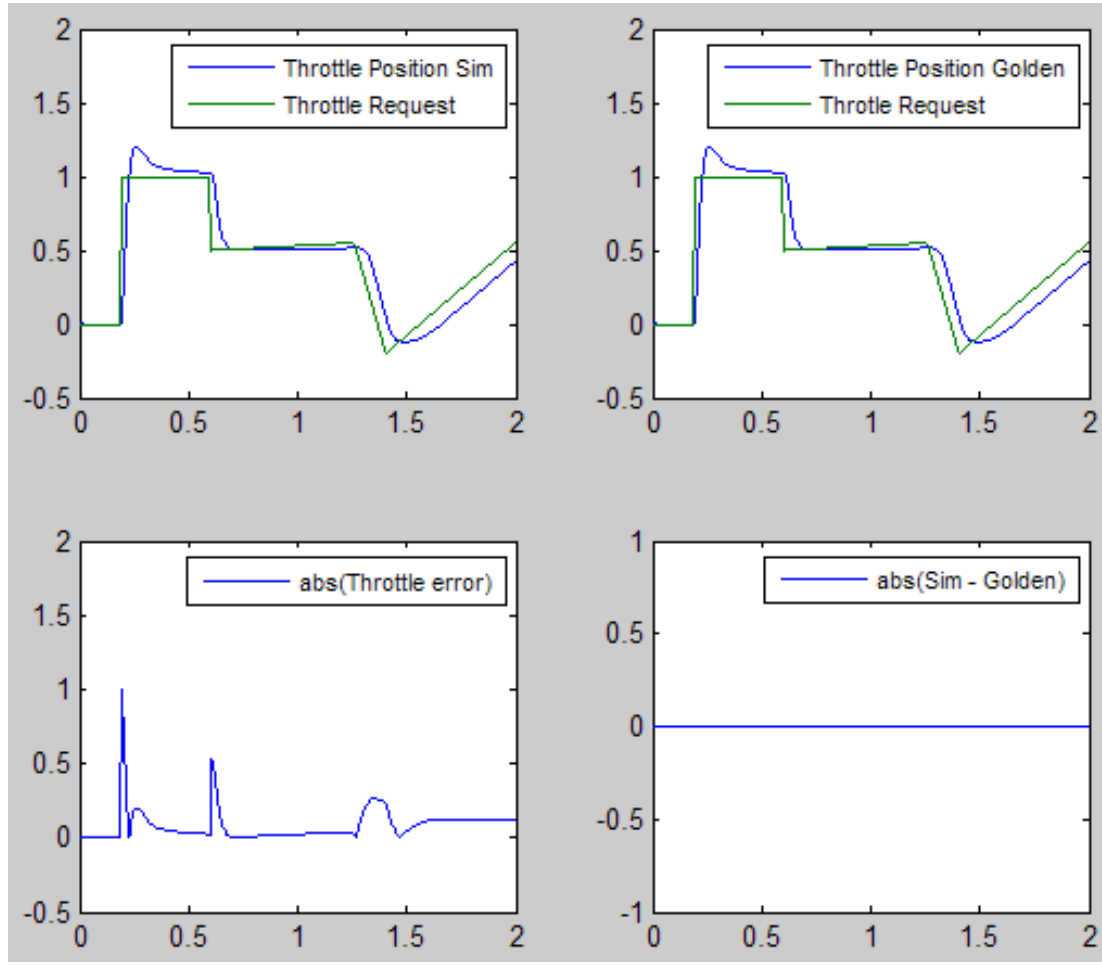
behavior when you run the generated code. You verify this effect by running the test harness.

Before you run your test harness, update it to include the `throttlecntrl_datainterface` model.

Note: The following procedure requires a Stateflow license.

1. Open `throttlecntrl_datainterface`.
2. Open your copy of test harness, `throttlecntrl_testharness`.
3. Right-click the `Unit_Under_Test` Model block and select **Block Parameters (ModelReference)**.
4. Set **Model name** to `throttlecntrl_datainterface`. Click **OK**.
5. Update the test harness model diagram.
6. Simulate the test harness.

   The resulting plot shows that the difference between the golden and simulated versions of the model remains zero.



7 Save and close `throttlecntrl_testharness`.

Manage Data

Data objects exist in a separate file from the model in the base workspace. To save the data manually, in the MATLAB Command Window, enter `save`.

The separation of data from the model provides the following benefits:

- One model, multiple data sets:
  - Use of different parameter values to change the behavior of the control algorithm (for example, for reusable components with different calibration values)
  - Use of different data types to change targeted hardware (for example, for floating-point and fixed-point targeted hardware)
- Multiple models, one data set:
  - Sharing data between models in a system
  - Sharing data between projects (for example, transmission, engine, and wheel controllers can use the same CAN message data set)

Key Points

- You can declare data in Simulink models and Stateflow charts by using data objects or direct specification.
- From the Model Explorer or from the command line in the MATLAB Command Window, manage (create, view, configure, and so on) base workspace data.
- The Data Object Wizard provides a quick way to create data objects for constructs such as signals, buses, and parameters.
- Configure data objects explicitly to appear by name in generated code.
- Separation of data from model provides several benefits.

More About

- "Load Signal Data for Simulation" (Simulink)
- "Data Representation" (Simulink Coder)
- "Custom Storage Classes"
- "Manage Placement of Data Definitions and Declarations" on page 36-100

Call External C Functions

In this section...

About This Example

Learning Objectives

- Evaluate a C function as part of a model simulation.
- Call an external C function from generated code.

Prerequisites

- Ability to open and modify Simulink models and subsystems.
- Ability to set model configuration parameters.
- Ability to read C code.
- An installed, supported C compiler.

Required Files

- `rtwdemo_throttlecntrl_extfunccall` model file
- `rtwdemo_ValidateLegacyCodeVrsSim` model file
- `/toolbox/rtw/rtwdemos/EmbeddedCoderOverview/stage_4_files/SimpleTable.c`
- `/toolbox/rtw/rtwdemos/EmbeddedCoderOverview/stage_4_files/SimpleTable.h`

Include External C Functions in a Model

Simulink models are one part of Model-Based Design. For many applications, a design also includes a set of pre-existing C functions created, tested (verified), and validated outside of a MATLAB and Simulink environment. You can integrate these functions easily into a model and the generated code. You can use external C code in the generated code to access hardware devices and external data files during rapid simulation runs.

This example shows you how to create a custom block that calls an external C function. When the block is part of the model, you can take advantage of the simulation environment to test the system further.

Create a Block That Calls a C Function

To specify a call to an external C function, use an S-Function block. You can automate the process of creating the S-Function block by using the Simulink Legacy Code Tool. Using this tool, specify an interface for your external C function. The tool then uses that interface to automate creation of an S-Function block.

1. Make copies of the files `SimpleTable.c` and `SimpleTable.h`, located in the folder *`matlabroot`*`/toolbox/rtw/rtwdemos/EmbeddedCoderOverview/stage_4_files` (open). Put the copies in your working folder.
2. Create an S-Function block that calls the specified function at each time step during simulation:

   a. In the MATLAB Command Window, create a function interface definition structure:

   ```
   def=legacy_code('initialize')
   ```

   The data structure `def` defines the function interface to the external C code.

   ```
   def =

                       SFunctionName: ''
       InitializeConditionsFcnSpec: ''
                      OutputFcnSpec: ''
                       StartFcnSpec: ''
                  TerminateFcnSpec: ''
                        HeaderFiles: {}
                        SourceFiles: {}
   ```

```
  HostLibFiles: {}
TargetLibFiles: {}
      IncPaths: {}
      SrcPaths: {}
      LibPaths: {}
    SampleTime: 'inherited'
       Options: [1x1 struct]
```

b Populate the function interface definition structure by entering the following commands:

```
def.OutputFcnSpec=['double y1 = SimpleTable(double u1,',...
   'double p1[], double p2[], int16 p3)'];
def.HeaderFiles = {'SimpleTable.h'};
def.SourceFiles = {'SimpleTable.c'};
def.SFunctionName = 'SimpTableWrap';
```

c Create the S-function:

```
legacy_code('sfcn_cmex_generate', def)
```

d Compile the S-function:

```
legacy_code('compile', def)
```

e Create the S-Function block:

```
legacy_code('slblock_generate', def)
```

A new model window opens that contains the `SimpTableWrap` block.

Tip: Creating the S-Function block is a one-time task. Once the block exists, you can reuse it in multiple models.

---

3 Save the model to your working folder as: `s_func_simptablewrap`.

4 Create a Target Language Compiler (TLC) file for the S-Function block:

```
legacy_code('sfcn_tlc_generate', def)
```

The TLC file is the component of an S-function that specifies how the code generator produces the code for a block.

For more information on using the Legacy Code Tool, see:

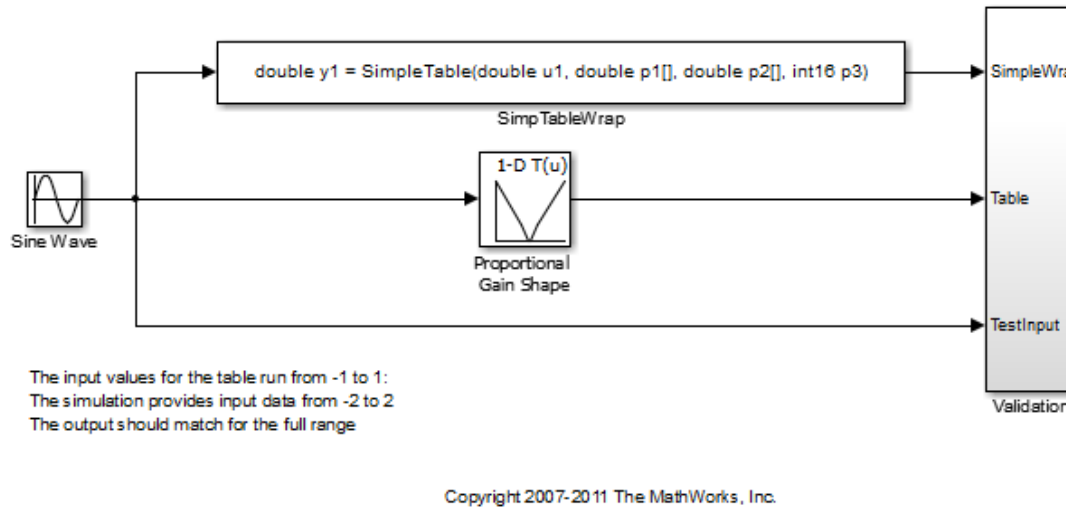
- "Integrate C Functions Using Legacy Code Tool" (Simulink)

- "Import Calls to External Code into Generated Code with Legacy Code Tool" (Simulink Coder)

Validate External Code in the Simulink Environment

When you integrate external C code with a Simulink model, before using the code, validate the functionality of the external C function code as a standalone component.

1 Open the model `rtwdemo_ValidateLegacyCodeVrsSim`. This model validates the S-function block that you created.

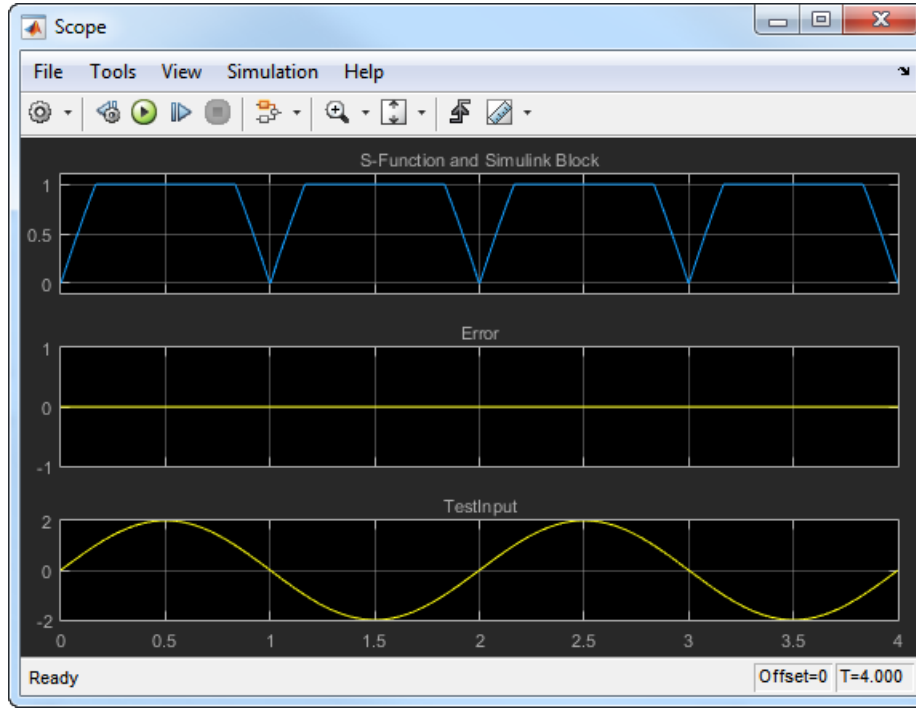


- The Sine Wave block produces output values from [-2 : 2].
- The input range of the lookup table is from [-1 : 1].
- The output from the lookup table is the absolute value of the input.
- The lookup table output clips the output at the input limits.

2 Simulate the model.

3 View the validation results by opening the `Validation` subsystem and, in that subsystem, clicking the Scope block.

The following figure shows the validation results. The external C code and the Simulink Lookup table block provide the same output values.



4 Close the validation model.

Validate C Code as Part of a Model

After you validate the functionality of the external C function code as a standalone component, validate the S-function in the model. Use the test harness model to complete the validation.

Note: The following procedure requires a Stateflow license.

1 Open `rtwdemo_throttlecntrl_extfunccall` and save a copy to `throttlecntrl_extfunccall` in a writable folder on your MATLAB path.

2 Examine the `PI_ctrl_1` and `PI_ctrl_2` subsystems.

a Lookup blocks have been replaced with the block you created using the Legacy Code Tool.

b Note the block parameter settings for `SimpTableWrap` and `SimpTableWrap1`.

c Close the Block Parameter dialog boxes and the PI subsystem windows.

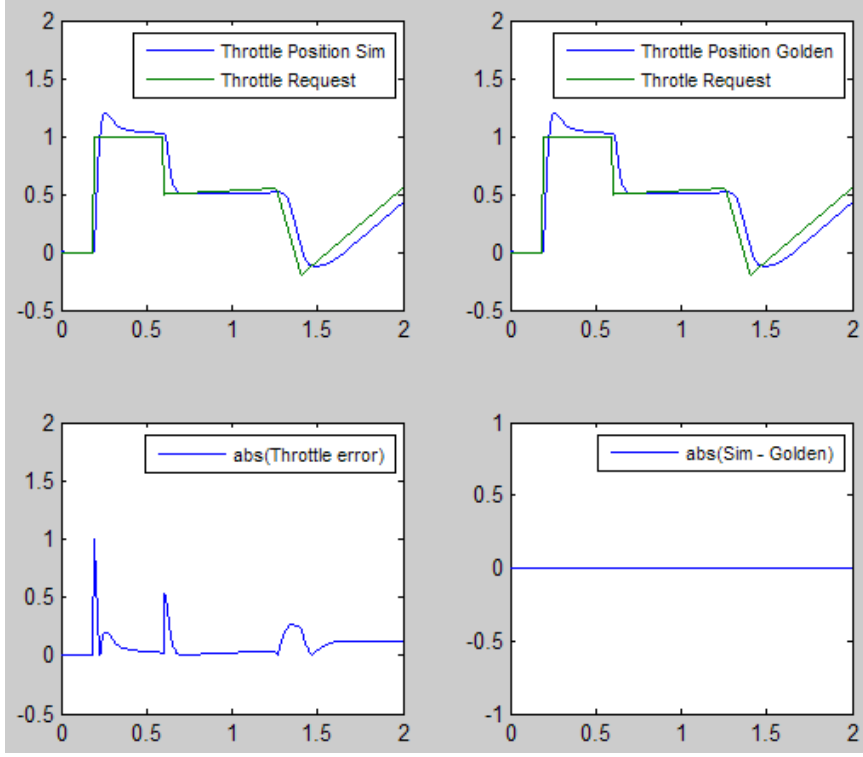
3 Open the test harness model, right-click the `Unit_Under_Test` Model block, and select **Block Parameters (ModelReference)**.

4 Set **Model name** to `throttlecntrl_extfunccall`. Click **OK**.

5 Update the test harness model diagram.

6 Simulate the test harness.

The simulation results match the expected golden values.



7 Save and close `throttlecntrl_extfunccall` and `throttlecntrl_testharness`.

Call a C Function from Generated Code

The code generator uses a TLC file to process the S-Function block. Calls to C code embedded in an S-Function block:

- Can use data objects.
- Are subject to *expression folding*, an operation that combines multiple computations into a single output calculation.

1 Open `throttlecntrl_extfunccall`.

2 Generate code for the model.

3 Examine the generated code in the file`throttlecntrl_extfunccall.c`.

4 Close `throttlecntrl_extfunccall` and `throttlecntrl_testharness`.

Key Points

- You can easily integrate external functions into a model and generated code by using the Legacy Code Tool.
- Validate the functionality of external C function code which you integrate into a model as a standalone component.
- After you validate the functionality of external C function code as a standalone component, validate the S-function in the model.

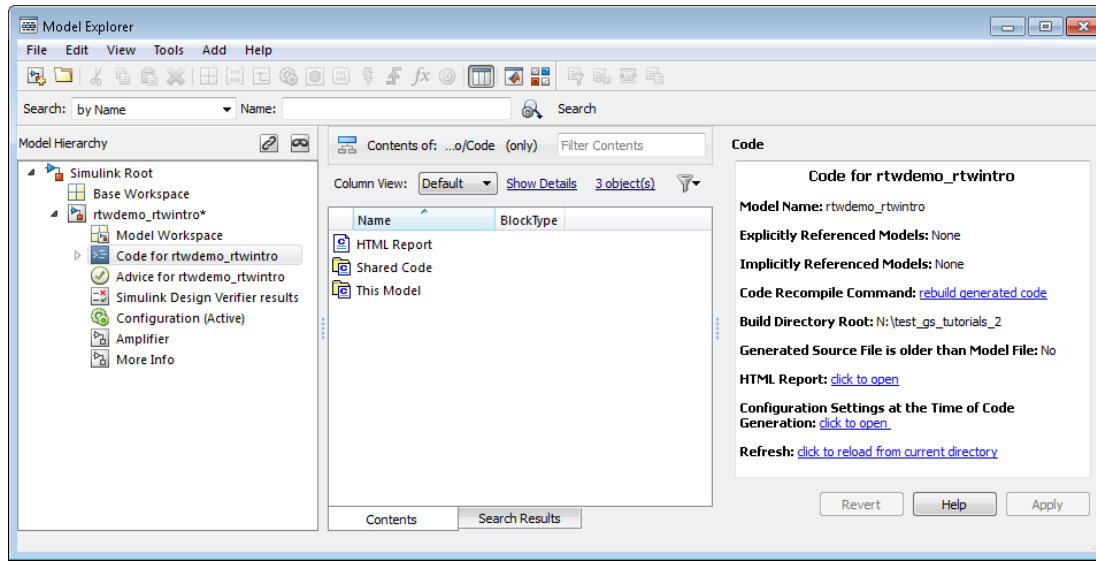
More About

- "Integrate C Functions Using Legacy Code Tool" (Simulink)
- "S-Functions and Code Generation" (Simulink Coder)

Reload Generated Code

You can reload the code generated for a model from the Model Explorer.

1. Click the **Code for *model*** node in the **Model Hierarchy** pane.
2. In the **Code** pane, click the **Refresh** link.



The code generator reloads the code for the model from the build folder.

More About

- "Rebuild a Model" (Simulink Coder)
- "Control Regeneration of Top Model Code" (Simulink Coder)

Manage Build Process Folders

By default, the build process places generated files from Simulink diagram updates and model builds in a hierarchy of folders. The code generation folder is the root folder of this folder hierarchy. The default code generation folder is the current working folder (`pwd`). If you are building models, artifacts used for simulation and code generation files reside in subfolders within that code generation folder. If you are building code for more than one referenced model within the same code generation folder, the model reference files are added to the existing `slprj` folder.

For specifying the folder locations, the software provides:

- MATLAB session parameters `CacheFolder` and `CodeGenFolder`
- Simulink preferences **Simulation cache folder** (Simulink) and **Code generation folder** (Simulink), which, if specified, provide the initial defaults for the MATLAB session parameters
- Function `Simulink.fileGenControl` for directly manipulating the MATLAB session parameters, for example, overriding or restoring the initial default values for the current session

For more information about managing build process folders, see these topics:

In this section...
“Select Simulation Cache Folder” on page 33-40
“Select Code Generation Folder” on page 33-40
“Override Build Folder Settings for Current Session” on page 33-41

For more information on organizing your files regarding code generation folders for referenced models, see “Generate Code for Referenced Models” (Simulink Coder).

The table lists folders where the build process places output files. The build folder contains the generated code modules *`model`*`.c`, *`model`*`.h`, and the generated makefile *`model`*`.mk`. In the paths in the table, *`model`* is the name of the model being used as a referenced model. And, *`target`* is the system target file acronym (for example, `grt`, `ert`, and `rsim`).

Folders	Description
“Code generation folder” (Simulink)	The default folder for code generation is the current working folder, `pwd`.

Folders	Description
	If you choose to generate an executable program file, the code generator writes the file *model*`.exe` (Windows) or *model* (UNIX) to your code generation folder. To choose a folder other than `pwd`, use Simulink preferences. See "Select Code Generation Folder" on page 33-40. When your default folder is set to the root folder of a drive, such as C:\, the code generator cannot generate code for your model.
"Simulation cache folder" (Simulink)	The default folder for simulation cache is the current working folder, `pwd`. If you choose to generate code for simulation, the code generator writes any model build artifacts for simulation to the simulation cache folder. To choose a folder other than `pwd`, use Simulink preferences. See "Select Simulation Cache Folder" on page 33-40.
model_target`_rtw`	The build folder — *model_target*`_rtw` A subfolder within your code generation folder. *model_target*`_rtw` is the name of the build folder. *model* is the name of the source model. The default for *target* is name of the selected system target file (for example, `grt` for the generic real-time system target file). You can change the *target* with the `rtwgensettings.BuildDirSuffix` field in the system target file. The build folder stores generated source code and other files created during the build process (except the executable program file).
model_target`_rtw/html`	The code generation report folder — *model_target*`_rtw/html` A subfolder within your build folder. *model_target*`_rtw/html` is the name of the code generation report folder. The report folder stores code generation report files created during the build process.

Folders	Description
`slprj`	Reference model code and shared utilities folder — `slprj` A subfolder within your "Code generation folder" (Simulink). When referenced models (model blocks) are built for simulation or code generation, the code generator places files in `slprj`. Subfolders in `slprj` provide separate places for simulation code, some generated code, utility code shared between models, and other files.
`slprj/sim/`*`model`*	Simulation target files for referenced models.
`slprj/sim/`*`model`*`/tmwinternal`	MAT-files used during code generation.
`slprj/sim/_sharedutils`	Utility functions for simulation system target files, shared across models.
`slprj/`*`target`*`/`*`model`*`/referenced_model_includes`	Header files from models referenced by this model.
`slprj/`*`target`*`/`*`model`*	Model reference target files.
`slprj/`*`target`*`/`*`model`*`/tmwinternal`	MAT-files used during code generation.
`slprj/`*`target`*`/_sharedutils`	Utility functions for model reference system target files, shared across models.

For an ERT-based system target file, the model configuration has additional shared utility options. For more information, see "Cross-Release Shared Utility Code Reuse" (Simulink Coder) and "Cross-Release Code Integration" (Simulink Coder).

- `UtilityFuncGeneration` parameter places generated shared code in `slprj` without the use of model reference.
- `ExistingSharedCode` parameter specifies the folder that contains previously generated shared code.
- `UseOnlyExistingSharedCode` parameter prohibits the build process from generating new shared code.
- `sharedCodeUpdate` function integrates shared code from multiple source folders into an existing shared code folder.

For more information about using code generation folders, see “Customize Build to Use Shared Utility Code” (Simulink Coder).

Select Simulation Cache Folder

By default, Simulink diagram updates place generated files in a build folder, the root of which is the current working folder (`pwd`). In some situations, you want the generated files to go to a root folder outside the current working folder. For example:

- You want to keep generated files separate from the models and other source materials that generate them.
- You want to reuse or share previously built simulation targets without having to set the current working folder back to a previous working folder.

The Simulink preference **Simulation cache folder** (Simulink) provides control over the output location for files generated by Simulink diagram updates. The preference appears in the **General** pane, under **File generation control**. To specify the root folder location for files generated by Simulink diagram updates, set the preference value by entering or browsing to a folder path. For example, on Windows, you could set the folder to `'C:\Work\mymodelsimcache'`.

The folder path that you specify provides the initial default for the MATLAB session parameter `CacheFolder`. When you initiate a Simulink diagram update, the update places the generated files in a build folder. This folder is at the root location specified by `CacheFolder` (if any), rather than in the current working folder (`pwd`).

You can choose to override the **Simulation cache folder** (Simulink) preference value for the current MATLAB session. See “Override Build Folder Settings for Current Session” on page 33-41.

Select Code Generation Folder

By default, Simulink model builds place generated files in a build folder, the root of which is the current working folder (`pwd`). Model builds potentially generate files for simulation and code generation system target files, and the resulting build folder contains artifacts for simulation and code generation. In some situations, you want the generated files to go to one or more root folders outside the current working folder. For example:

- You want to keep generated files separate from the models and other source materials that generate them.

- You want to separate generated production code from generated simulation artifacts.

The Simulink preference **Code generation folder** (Simulink) provides control over the output location for files that model builds generate for system target files. The preference appears in the **General** pane, under **File generation control**. To specify the root folder location for code generation files generated by model builds, set the preference value by entering or browsing to a folder path. For example, on Windows, you could set the **Code generation folder** (Simulink) to `'C:\test\mymodelgencode'`.

The folder path that you specify provides the initial default for the MATLAB session parameter `CodeGenFolder`. When you initiate a Simulink model build, the build process places the generated files in a build folder. This folder is at the root location specified by `CodeGenFolder` (if any), rather than in the current working folder (`pwd`).

You can choose to override the **Code generation folder** (Simulink) preference value for the current MATLAB session. See "Override Build Folder Settings for Current Session" on page 33-41.

Override Build Folder Settings for Current Session

The Simulink preferences **Simulation cache folder** (Simulink) and **Code generation folder** (Simulink) provide initial defaults for the MATLAB session parameters `CacheFolder` and `CodeGenFolder`. These session parameters determine where Simulink diagram updates and model builds place generated files.

You can override these build folder settings during the current MATLAB session by using the `Simulink.fileGenControl` function. With this function, you can manipulate directly the MATLAB session parameters, for example, overriding or restoring the initial default values. The values that you set by using `Simulink.fileGenControl` expire at the end of the current MATLAB session. For more information and detailed examples, see `Simulink.fileGenControl`.

More About

- "Manage Build Process Files" (Simulink Coder)
- "Manage Build Process File Dependencies" (Simulink Coder)
- "Add Build Process Dependencies" (Simulink Coder)
- "Enable Build Process for Folder Names with Spaces" (Simulink Coder)
- "Build and Run a Program" (Simulink Coder)

Manage Build Process Files

The code generator creates *model*.* files during the code generation and build process. The code generator creates additional folders and dependency files to support shared utilities and model references.

For more information about the folders that the build process creates, see "Manage Build Process Folders" (Simulink Coder).

The source and header files in the table have dependency relationships. For descriptions of other file dependencies, see "Manage Build Process File Dependencies" (Simulink Coder) and "Add Build Process Dependencies" (Simulink Coder).

Depending on model architectures and code generation options, the build process for a GRT-based system target file can produce files that the build process does not generate for an ERT-based system target file. Also, for ERT-based system target files, the build process packages generated files differently than for GRT-based system target files. See "Manage File Packaging of Generated Code Modules" on page 34-14.

The table describes the principal generated files. Within the generated file names shown in the table, the *model* represents the name of the model for which you are generating code. The *subsystem* represents the name of a subsystem within the model. When you select the **Create code generation report** parameter, the code generator produces a set of HTML files. There is one HTML file for each source file plus a *model*_contents.html index file in the html subfolder within your build folder.

File	Description
builtin_typeid_types.h	Defines an enumerated type corresponding to built-in data types. A model build generates this file when one or more of these conditions apply: • Your model contains a Stateflow chart that uses messages. • Your model configuration enables **MAT-file logging**. • Your model configuration enables C API options in **Code Generation** > **Interface**.
modelsources.txt	Lists additional sources to include in the compilation.
model.bat	Contains Windows batch file commands that set the compiler environment and invoke the make utility.

File	Description
	For more information about using this file, see “model.bat” on page 33-48.
model.c *model*.cpp	Corresponds to the model file. The Target Language Compiler generates this C or C++ source code file. The file contains: • Include files *model*.h and *model*_private.h • Data, except data placed in *model*_data.c • Model-specific scheduler code • Model-specific solver code • Model registration code • Algorithm code • Optional GRT wrapper functions
model.exe (Windows platform) *model* (UNIX and Macintosh platforms)	Executable program file. A model build generates this file unless you explicitly specify that the code generator produce code only. The build generates the executable in the current folder (not the build folder) under control of the make utility of your development system.
model.h	Defines model data structures and a public interface to the model entry points and data structures. Provides an interface to the real-time model data structure (*model*_rtM) via access macros. Subsystem .c or .cpp files in the model include *model*.h. This file includes: • Exported Simulink data symbols • Exported Stateflow machine parented data • Model data structures, including rtM • Model entry-point functions For more information, see “model.h” (Simulink Coder).

File	Description
model.mk	Generated makefile that controls compiling and linking the generated code into the final binary file by the `make` utility of your development system. If you set the `MAKEFLAGS` environment variable, do not select options with this variable that conflict with the current `make` utility used by the build process.
model.rtw	Represents the compiled model. By default, the build process deletes this ASCII file when the build process is complete. You can choose to retain the file for inspection.
*model*_capi.h *model*_capi.c	(optional files) Contain data structures that describe the model signals, states, and parameters without using external mode. For more information, see "Exchange Data Between Generated and External Code Using C API" (Simulink Coder).
*model*_data.c	Contains (if conditionally generated) declarations for the parameters data structure and the constant block I/O data structure, and zero representations for structure data types that the model uses. A model build generates this file when the model uses these data structures. The `extern` declarations for structures appear in *model*.h. When present, this file contains: • Constant block I/O parameters • Include files *model*.h and *model*_private.h • Definitions for the zero representations for user-defined structure data types that the model uses • Constant parameters
*model*_dt.h	(optional file) Declares structures that contain data type and data type transition information for generated model data structures for supporting external mode.

File	Description
*model*_private.h	Contains local `define` constants and local data for the model and subsystems. The generated source files from the model build include this file. When you interface external code with generated code from a model, include *model*_private.h. The file contains: • Imported Simulink data symbols • Imported Stateflow machine parented data • Stateflow entry points • Simulink Coder details (various macros, `enum`s, and so forth, that are private to the code) For more information, see "Manage Build Process File Dependencies" (Simulink Coder).
model_reference_types.h	Contains type definitions for timing bridges. A model build generates this file for a referenced model or a model containing model reference blocks.
*model*_targ_data_map.m	(optional file) Contains MATLAB language commands that external mode uses to initialize the external mode connection.
*model*_types.h	Provides forward declarations for the real-time model data structure and the parameters data structure. The generated header files from the model build include this file. Function declarations of reusable functions can use these structures.

File	Description
`multiword_types.h`	Contains type definitions for multiple-word wide data types and their word-size chunks. If your code uses multiword data types, include this header file. A model build generates this file when one or more of these conditions apply: • Your model uses multiword data types. • Your model configuration enables **MAT-file logging**. • Your model configuration enables **Code Generation** > **Interface** > **External mode**.

`rtGetInf.c`
 	
`rtGetInf.h`
 	
`rtGetNaN.c`
 	
`rtGetNaN.h`
 	
`rt_nonfinite.c`
 	
`rt_nonfinite.h`
	Declares and initializes global nonfinite values for `inf`, minus `inf`, and `nan`. Provides nonfinite comparison functions.
	
	A model build generates these files when one or more of these conditions apply:
	
	• The model contains S-functions.
	• The generated code from the model requires nonfinite numbers.
	• Your model configuration enables **MAT-file logging**.
	• Your model configuration selects `grt.tlc` as the **System target file** and enables the **Classic call interface**.

`rtmodel.h`	Contains `#include` directives required by static main program modules such as `rt_main.c`. The build process does not create these modules at code generation time. The modules include `rt_model.h` to access model-specific data structures and entry points. If you create your own main program module, make sure to include `rtmodel.h`.

File	Description
`rtwtypes.h`	Provides the essential type definitions, `#define` statements, and enumerations. For GRT-based system target files, `rtwtypes.h` includes `simstruc_types.h` which, in turn, includes `tmwtypes.h`. For ERT-based system target files that do not generate a GRT interface and do not have noninlined S-functions, `rtwtypes.h` does not include `simstruc_types.h`. For more information, see "rtwtypes.h" (Simulink Coder) and "Manage Build Process File Dependencies" (Simulink Coder).

`rtw_proj.tmw`
 	
`sl_proj.tmw`
	Marker files.
	
	The build process generates these files to help the `make` utility determine when to recompile and link the generated code.

`rt_defines.h`	Contains type definitions for special mathematical constants (such as π and *e*) and defines the `UNUSED_PARAMETER` macro. A model build generates this file when the generated code requires a mathematical constant definition or when the function body does not access a required model function argument.

`rt_sfcn_helper.h`
 	
`rt_sfcn_helper.c`
	(optional files) Provide functions that the noninlined S-functions use in a model.
	
	The noninlined S-functions use functions `rt_CallSys`, `rt_enableSys`, and `rt_DisableSys` to call downstream function-call subsystems.

`subsystem``.c`	(optional file) Contains C source code for each noninlined nonvirtual subsystem or copy the code when the subsystem is configured to place code in a separate file.
`subsystem``.h`	(optional file) Contains exported symbols for noninlined nonvirtual subsystems.

model.bat

This file contains Windows batch file commands that set the compiler environment and invoke the `make` utility.

If you are using the toolchain approach for the build process, you also can use this batch file to extract information from the generated makefile, *model*`.mk`. The information includes macro definitions and values that appear in the makefile, such as `CFLAGS` (C compiler flags) and `CPP_FLAGS` (C++ compiler flags). With the folder containing *model*`.bat` selected as the current working folder, in the Command Window, type:

```
>> system('model.bat info')
```

On UNIX and Macintosh platforms, the code generator does not create the *model*`.bat` file. To extract information for toolchain approach builds from the generated makefile on these systems, in the Command Window, type:

```
>> system('gmake –f model.mk info')
```

model.h

The header file *model*`.h` declares model data structures and a public interface to the model entry points and data structures. This header file also provides an interface to the real-time model data structure (*model*`_M`) by using access macros. If your code interfaces to model functions or model data structures, include *model*`.h`:

- Exported global signals

  ```
  extern int32_T INPUT;    /* '<Root>/In' */
  ```

- Global structure definitions

  ```
  /* Block parameters (auto storage) */
  extern Parameters_mymodel mymodel_P;
  ```

- Real-time model (RTM) macro definitions

  ```
  #ifndef rtmGetSampleTime
  # define rtmGetSampleTime(rtm, idx)
  ((rtm)->Timing.sampleTimes[idx])
  #endif
  ```

- Model entry point functions (ERT example)

  ```
  extern void mymodel_initialize(void);
  ```

```
extern void mymodel_step(void);
extern void mymodel_terminate(void);
```

The `main.c` (or `.cpp`) file includes *`model`*`.h`. If the model build generates the `main.c` (or `.cpp`) file from a TLC script, the TLC source can include *`model`*`.h`.

```
#include "%<CompiledModel.Name>.h"
```

If `main.c` is a static source file, you can use the fixed header file name `rtmodel.h`. This file includes the *`model`*`.h` header file:

```
#include "model.h"     /* If main.c is generated */
```

or

```
#include "rtmodel.h"   /* If static main.c is used */
```

Other external source files can require to include *`model`*`.h` to interface to model data, for example exported global parameters or signals. The *`model`*`.h` file itself can have additional header dependencies due to requirements of generated code. See "System Header Files" (Simulink Coder) and "Code Generator Header Files" (Simulink Coder).

To reduce dependencies and reduce the number of included header files, see "Manage Build Process File Dependencies" (Simulink Coder).

rtwtypes.h

The header file `rtwtypes.h` defines data types, structures, and macros required by the generated code. You include `rtwtypes.h` for GRT and ERT system target files, instead of including `tmwtypes.h` or `simstruc_types.h`.

Often, the generated code requires that integer operations overflow or underflow at specific values. For example, when the code expects a 16-bit integer, the code does not accept an 8-bit or a 32-bit integer type. The C language does not set a standard for the number of bits in types such as `char`, `int`, and others. So, there is no universally accepted data type in C to use for sized-integers.

To accommodate this feature of the C language, the generated code uses sized integer types, such as `int8_T`, `uint32_T`, and others, which are not standard C types. In `rtwtypes.h`, the generated code maps these sized-integer types to the corresponding C keyword base type using information in the **Hardware Implementation** pane of the configuration parameters.

The code generator produces the optimized version of `rtwtypes.h` for ERT-based system target files when these conditions exist:

- The model has a cleared **Classic call interface** option in the **All Parameters** tab of the Configuration Parameters dialog box.
- The model does not contain noninlined S-functions.

Include `rtwtypes.h`. If you include it for GRT system target files, for example, it is easier to use your code with ERT-based system target files.

For GRT and ERT system target files, the location of `rtwtypes.h` depends on whether the build process uses the *shared utilities* location. If it uses a shared location, the code generator places `rtwtypes.h` in `slprj/target/_sharedutils`; otherwise, it places `rtwtypes.h` in the build folder (*`model_target`*`_rtw`). See "Specify Generated Code Interfaces" (Simulink Coder).

Source files include the `rtwtypes.h` header file when the source files use code generator type names or other code generator definitions. A typical example is for files that declare variables by using a code generator data type, for example, `uint32_T myvar`.

A source file that the code generator and an S-function use can use the preprocessor macro `MATLAB_MEX_FILE`. The macro definition comes from the `mex` function:

```
#ifdef  MATLAB_MEX_FILE
#include "tmwtypes.h"
#else
#include "rtwtypes.h"
#endif
```

A source file for the code generator `main.c` (or `.cpp`) file includes `rtwtypes.h` without preprocessor checks.

```
#include "rtwtypes.h"
```

Custom source files that the Target Language Compiler generates can also emit these `include` statements into their generated file.

See "Control Placement of rtwtypes.h for Shared Utility Code" (Simulink Coder).

More About

- "Manage Build Process Folders" (Simulink Coder)
- "Manage Build Process File Dependencies" (Simulink Coder)

- “Add Build Process Dependencies” (Simulink Coder)
- “Enable Build Process for Folder Names with Spaces” (Simulink Coder)
- “Build and Run a Program” (Simulink Coder)

Manage Build Process File Dependencies

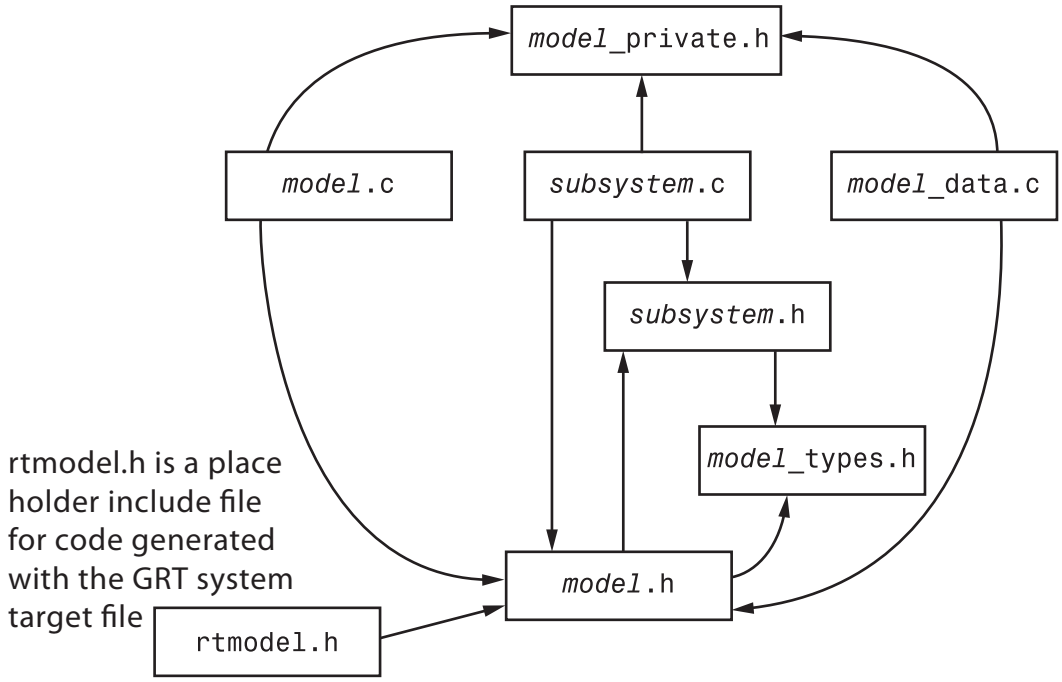
The dependency relationships among generated source and header files appear in the figure. Arrows coming from a file point to files it includes. Other dependencies exist, for example, on Simulink header files `tmwtypes.h` and `simstruc_types.h`, plus C or C ++ library files. The figure maps inclusion relations between only those files that are generated in the build folder. These files can reference utility and model reference code located in a code generation folder. For more information about the folders and files that the build process creates, see "Manage Build Process Folders" (Simulink Coder) and "Manage Build Process Files" (Simulink Coder).

The two tables identify the conditions that control creation of dependency files for GRT and ERT targets. To manage build-related dependencies, consider how these conditions apply to your model and code generation process. Then, configure model parameters and code generation options to manage build process file dependencies.

Due to differences in file packaging options for code generated with ERT-based system target files, the file dependencies differ slightly from file packaging for code generated with GRT-based system target files. See "Manage File Packaging of Generated Code Modules" on page 34-14.

The parent system header files (*`model`*`.h`) include child subsystem header files (*`subsystem`*`.h`). In more layered models, subsystems similarly include their children's header files in the model hierarchy. As a consequence, subsystems are able to view recursively into their descendant subsystems and view into the root system because every *`subsystem`*`.c` or *`subsystem`*`.cpp` includes *`model`*`.h` and *`model`*`_private.h`.

In the figure, files *`model`*`.h`, *`model`*`_private.h`, and *`subsystem`*`.h` depend on the header file `rtwtypes.h`. If you use system target files that are not based on the ERT system target file, the source files that you generate can have additional dependencies on `tmwtypes.h` and `simstruc_types.h`.



For information about file dependencies in header files, see the following:

In this section...
“System Header Files” on page 33-53
“Code Generator Header Files” on page 33-56

System Header Files

The system header files make function declarations, type definitions, and macro definitions available to the legacy or external code. Some code generation scenarios require including header files that are specific to the code generator product.

The code generator includes some system header files for broadly defined cases. For example, generated code includes `<stddef.h>` when the model contains a utility function that requires this header file. This approach helps identify header file dependencies:

1. Set the Shared code placement parameter to `'Shared location'` and build the model. The code generator places the utility functions in `__sharedutils` folder.
2. Use a find-in-file utility (for example, grep utility) to search the `.c` and `.h` files in the `__sharedutils` folder for `#include`. The search results list the utilities with header file dependencies.
3. Use this information to identify utilities to remove from the model and reduce header file dependencies in the generated code.

For more information, see "Generate Shared Utility Code for Fixed-Point Functions" (Simulink Coder).

System Header File	Description and Inclusion Conditions for GRT or ERT System Target Files
`<math.h>`	Defines math constants GRT—Generated code does not include this file. ERT—Generated code includes this file when the code honors your model configuration for solver **Stop time** and either: • Your model configuration enables **MAT-file logging**. See "MAT-file logging" (Simulink Coder). • Your model configuration enables **Code Generation** > **Interface** > **External mode**.
`<float.h>`	Provides floating-point math functions GRT—Generated code includes this file when your model contains a floating-point math function. ERT—Generated code includes this file when your model contains a floating-point math function, unless a code replacement library entry overrides the function. For more information, see "Choose a Code Replacement Library" (Simulink Coder).
`<stddef.h>`	Defines NULL GRT and ERT—Generated code includes this file when your model contains a utility function that requires this file.
`<stdio.h>`	Provides file I/O functions

System Header File	Description and Inclusion Conditions for GRT or ERT System Target Files
	GRT—Generated code includes this file when your model includes a To File block. ERT—Generated code includes this file when either: • Your model includes a To File block. • Your model configuration enables **MAT-file logging**. See "MAT-file logging" (Simulink Coder).
`<stdlib.h>`	Provides utility functions such as the integer versions of `div()` and `abs()` GRT—Generated code includes this file when either: • Your model includes a Stateflow chart. • Your model includes a math function block configured for `mod()` or `rem()`, which generate calls to `div()`. ERT—Generated code includes this file when either: • Your model includes a Stateflow chart, and you select **Support: floating-point numbers**. • Your model includes a math function block configured for `mod()` or `rem()`, which generate calls to `div()`.
`<string.h>`	Provides memory functions such as `memset()` and `memcpy()` GRT—Generated code includes this file when your model initialization code calls `memset()`. ERT—Generated code includes this file when a block or model initialization code calls `memcpy()` or `memset()`. For a list of relevant blocks, in the Command Window, type: `showblockdatatypetable` Look for blocks with the N2 note. To omit calls to `memset()` from model initialization code, select the **Remove root level I/O zero initialization** and **Remove internal data zero initialization** optimization configuration parameters.

Code Generator Header Files

Dependencies in the table for generated header files apply to the system target files `grt.tlc` and `ert.tlc`. System target files derived from these base system target files can have additional header dependencies. Code generation for blocks from blocksets, embedded targets, and custom S-functions can introduce additional header dependencies.

Header File	Description and Inclusion Conditions for GRT or ERT System Target Files
`builtin_typeid_types.h`	Defines an enumerated type corresponding to built-in data types GRT and ERT—Generated code includes this file when one or more of these conditions apply: • Your model contains a Stateflow chart that uses messages. • Your model configuration enables: **MAT-file logging**. See "MAT-file logging" (Simulink Coder). • Your model configuration selects C API options at **Code Generation** > **Interface**.
`dt_info.h`	Defines data structures for external mode GRT and ERT—Generated code includes this file when your model configuration enables external mode.
`ext_work.h`	Defines external mode functions GRT and ERT—Generated code includes this file when your model configuration enables external mode.
`fixedpoint.h`	Provides fixed-point support for noninlined S-functions GRT—Generated code includes this file. ERT—Generated code includes this file when either: • Your model uses noninlined S-functions. • Your model configuration selects **Classic call interface**.
`model_reference_types.h`	Contains type definitions for timing bridges GRT and ERT—Generated code includes this file when building a reference model or building a model that contains model blocks.

Header File	Description and Inclusion Conditions for GRT or ERT System Target Files
*model*_types.h	Defines model-specific data types GRT and ERT—Generated code includes this file.
multiword_types.h	Contains type definitions for multiword-wide data types and their word-size chunks GRT and ERT—Generated code includes this file when one or more of these conditions apply: • Your model uses multiword data types. • Your model configuration enables **MAT-file logging**. See "MAT-file logging" (Simulink Coder). • Your model configuration enables **Code Generation** > **Interface** > **External mode**.
rtGetInf.h rtGetNaN.h rt_nonfinite.h	Support nonfinite numbers GRT—Generated code includes this file when one or more of these conditions apply: • Your model contains S-functions. • The generated code requires nonfinite numbers. • Your model configuration enables **MAT-file logging**. See "MAT-file logging" (Simulink Coder). • Your model configuration selects the **Classic call interface**. ERT—Generated code includes this file when one or more of these conditions apply: • Your model contains S-functions. • The generated code requires nonfinite numbers. • Your model configuration enables **MAT-file logging**. See "MAT-file logging" (Simulink Coder).

Header File	Description and Inclusion Conditions for GRT or ERT System Target Files
`rt_defines.h`	Contains type definitions for special mathematical constants (such as π and *e*) and defines the `UNUSED_PARAMETER` macro GRT and ERT—Generated code includes this file when either: • The generated code requires a mathematical constant definition. • The function body does not access a required model function argument.

`rt_logging.h`	Supports MAT-file logging and includes:
	
	`rtwtypes.h`
	`builtin_typeid_types.h`
	`multiword_types.h`
	`rt_mxclassid.h`
	`rtw_matlogging.h`
	
	GRT—Generated code includes this file.
	
	ERT—Generated code includes this file when you model configuration enables **MAT-file logging**. See "MAT-file logging" (Simulink Coder).

`rt_mxclassid.h`	Defines `mxArray` class ID enumerations GRT and ERT—Generated code includes this file when the code includes `rt_logging.c`.
`rtw_continuous.h`	Supports continuous time GRT—Generated code includes this file when the code includes `simstruc_types.h`. ERT—Generated code includes this file when your model configuration selects **Support: continuous time** and when the code does not already include `simstruc.h`.

Header File	Description and Inclusion Conditions for GRT or ERT System Target Files
`rtw_extmode.h`	Supports external mode GRT—Generated code includes this file when the code includes `simstruc_types.h`. ERT—Generated code includes this file when your model configuration selects external mode and when the code does not already include `simstruc.h`.
`rtw_matlogging.h`	Supports MAT-file logging GRT—Generated code includes this file when the code includes `simstruc_types.h` and `rt_logging.h`. ERT—Generated code includes this file when the code includes `rt_logging.h`.
`rtw_solver.h`	Supports continuous states GRT—Generated code includes this file when the code includes `simstruc_types.h`. ERT—Generated code includes this file when your model configuration selects **Support: continuous time** and when the code does not already include `simstruc.h`.
`rtwtypes.h`	Defines code generator data types GRT—Generated code includes this file. Use the complete version of the file, which includes `tmwtypes.h` and `simstruc_types.h`. See `simstruc_types.h` for dependencies. ERT—Generated code includes this file. Use the complete or optimized version of the file. See “rtwtypes.h” on page 33-49. If you include `rtwtypes.h` and `tmwtypes.h` in external code that you integrate with generated code, the `#include` for `rtwtypes.h` must precede the `#include` for `tmwtypes.h`. This ordering of `#include` statements preserves target-specific type definitions. If you reverse the order, a compiler error occurs.

Header File	Description and Inclusion Conditions for GRT or ERT System Target Files

`simstruc.h`	Supports calling noninlined S-functions that use the `Simstruct` definition; also includes:
	
	`limits.h`
	`string.h`
	`tmwtypes.h`
	`simstruc_types.h`
	
	GRT—Generated code includes this file.
	
	ERT—Generated code includes this file when either:
	
	• Your model uses noninlined S-functions.
	• Your model configuration selects **Classic call interface**.

`simstruc_types.h`	Provides definitions that the generated code uses and includes the header files:
	
	`rtw_matlogging.h`
	`rtw_extmode.h`
	`rtw_continuous.h`
	`rtw_solver.h`
	`sysran_types.h`
	
	GRT—Generated code includes this file when the code includes `rtwtypes.h`.
	
	ERT—Generated code does not include this file. For ERT, `rtwtypes.h` contains definitions, and *`model`*`.h` contains header files.

`sysran_types.h`	Supports external mode GRT—Generated code includes this file when the code includes `simstruc_types.h`. ERT—Generated code includes this file when your model configuration selects external mode and when the code does not already include `simstruc.h`.

More About

- "Manage Build Process Folders" (Simulink Coder)
- "Manage Build Process Files" (Simulink Coder)
- "Add Build Process Dependencies" (Simulink Coder)
- "Enable Build Process for Folder Names with Spaces" (Simulink Coder)
- "Build and Run a Program" (Simulink Coder)

Add Build Process Dependencies

When you specify a system target file for code generation, the code generator can build a standalone executable program that can run on the development computer. To build the executable program, the code generator uses the selected compiler and the generated makefile for a toolchain approach or for a template makefile (TMF) approach build processes. Part of the makefile generation process is to add source file, header file, and library file information (the dependencies) in the generated makefile for a compilation. Or, for a specific application, you can add the generated files and file dependencies through a configuration management system.

The generated code for a model consists of a small set of files. (See "Manage Build Process Files" (Simulink Coder).) These files have dependencies on other files, which occur due to:

- Header file inclusions
- Macro declarations
- Function calls
- Variable declarations

The model or external code introduces dependencies for various reasons:

- Blocks in a model generate code that makes function calls. These calls can occur in several forms:
  - Included source files (not generated) declare the called functions. In cases such as a blockset, manage these source file dependencies by compiling them into a library file.
  - The generated code makes calls to functions in the run-time library provided by the compiler.
  - Some function dependencies also are generated files, which are referred to as shared utilities. Some examples are fixed-point utilities and non-finite support functions. These dependencies are referred to as shared utilities. The generated functions can appear in files in the build folder for standalone models or in the `_sharedutils` folder under the `slprj` folder for builds that involve model reference.
- Models with continuous time require solver source code files.
- Code generator options such as external mode, C API, and MAT-file logging.
- External code specifies dependencies.

For information about adding file dependencies in information for the build process, see the following:

In this section...

File Dependency Information for the Build Process

The code generator provides several mechanisms to input file dependency information into the build process. The mechanisms depend on whether your dependencies are block-based or are model- or system target file-based.

For block dependencies, consider using:

- S-functions and blocksets
  - Add folders that contain S-function MEX-files that the model uses to the header include path.
  - Create makefile rules for these folders to allow for finding source code.
  - Specify additional source file names with the S-Function block parameter `SFunctionModules`.
  - Specify additional dependencies with the `rtwmakecfg.m` mechanism. See "Use rtwmakecfg.m API to Customize Generated Makefiles" (Simulink Coder).

  For more information on applying these approaches to legacy or external code integration, see "Import Calls to External Code into Generated Code with Legacy Code Tool" (Simulink Coder).
- S-Function Builder block, which provides its own UI for specifying dependency information

For model- or system target file-based dependencies, such as external header files, consider using:

- The **Code Generation** > **Custom Code** pane in the Configuration Parameters dialog box. You can specify additional libraries, source files, and include folders.
- TLC functions `LibAddToCommonIncludes()` and `LibAddToModelSources()`. You can specify dependencies during the TLC phase. See "LibAddToCommonIncludes(incFileName)" (Simulink Coder) and

"LibAddSourceFileCustomSection(file, builtInSection, newSection)" (Simulink Coder). The Embedded Coder product also provides a TLC-based customization template for generating additional source files.

Generated Makefile Dependencies

For toolchain approach or template makefile (TMF) approach build processes, the code generator generates a makefile. For TMFs, the generated makefile provides token expansion in which the build process expands different tokens in the makefile to include the additional dependency information. The resulting makefile contains the complete dependency information. See "Customize Template Makefiles" (Simulink Coder).

The generated makefile contains:

- Names of the source file dependencies
- Folders where source files are located
- Location of the header files
- Precompiled library dependencies
- Libraries that the `make` utility compiles and creates

A property of `make` utilities is that you do not have to specify the specific location for a given source C or C++ file. If a rule exists for that folder and the source file name is a prerequisite in the makefile, the make utility can find the source file and compile it. The C or C++ compiler (preprocessor) does not require absolute paths to the headers. The compiler finds header file with the name of the header file by using an `#include` directive and an include path. The generated C or C++ source code depends on this standard compiler capability.

Libraries are created and linked against, but occlude the specific functions that the program calls.

These properties can make it difficult to determine the minimum list of file dependencies manually. You can use the makefile as a starting point to determine the dependencies in the generated code. For an example that shows how to identify dependencies, see "Relocate Code to Another Development Environment with packNGo" (Simulink Coder).

Another approach to determining the dependencies is using linker information, such as a linker map file, to determine the symbol dependencies. The map file provides the location of code generator and blockset source and header files to help in locating the dependencies.

Code Generator Static File Dependencies

Several locations in the MATLAB folder tree contain static file dependencies specific to the code generator:

- *matlabroot*`/rtw/c/src` (open)

  This folder has subfolders and contains additional files must be compiled. Examples include solver functions (for continuous time support), external mode support files, C API support files, and S-function support files. Include source files in this folder into the build process with the `SRC` variables of the makefile.
- Header files in the folder `matlabroot/rtw/extern/include`
- Header files in the folder `matlabroot/simulink/include`

  These folders contain additional header file dependencies such as `tmwtypes.h`, `simstruc_types.h`, and `simstruc.h`.

  Note For ERT-based system target files, you can avoid several header dependencies. ERT-based system target files generate the minimum set of type definitions, macros, and so on, in the file `rtwtypes.h`.

Blockset Static File Dependencies

Blockset products with S-function code apply the `rtwmakecfg.m` mechanism to provide the code generator with dependency information. The `rtwmakecfg.m` file from the blockset contains the list of include path and source path dependencies for the blockset. Typically, blocksets create a library from the source files to which the generated model code can link. The libraries are created and identified when you use the `rtwmakecfg.m` mechanism.

To locate the `rtwmakecfg.m` files for blocksets in your MATLAB installed tree, use the following command:

```
>> which -all rtwmakecfg.m
```

If the model that you are compiling uses one or more of the blocksets listed by the `which` command, you can determine folder and file dependency information from the respective `rtwmakecfg.m` file.

Folder Dependency Information for the Build Process

You can add `#include` statements to generated code. Such references can come from several sources, including TLC scripts for inlining S-functions, custom storage classes, bus objects, and data type objects. The included files consist of header files for external code or other customizations. You can specify compiler include paths with the `-I` compiler option. The build process uses the specified paths to search for included header files.

Usage scenarios for the generated code include, but are not limited to, the following:

- A custom build process compiles generated code that requires an environment-specific set of `#include` statements.

  In this scenario, the build process invokes the code generator when you select the **Generate code only** check box. Consider using fully qualified paths, relative paths, or just the header file names in the `#include` statements. Use include paths.

- The build process compiles the generated code.

  In this case, you can specify compiler include paths (`-I`) for the build process in several ways:

  - Specify additional include paths on the **Code Generation** > **Custom Code** pane in the Configuration Parameters dialog box. The code generator propagates the include paths into the generated makefile.
  - The `rtwmakecfg.m` mechanism allows S-functions to introduce additional include paths into the build process. The code generator propagates the include paths into the generated makefile.
  - When building a model that uses a custom system target file and is makefile-based, you can directly add the include paths into the template makefile that the system target file uses.
  - Use the `make` command to specify a `USER_INCLUDES` make variable that defines a folder in which the build process searches for included files. For example:

    ```
    make_rtw USER_INCLUDES=-Id:\work\feature1
    ```

    The build process passes the custom includes to the command-line invocation of the make utility, which adds them to the overall flags passed to the compiler.

Use `#include` Statements and Include Paths

Consider the following approaches for using `#include` statements and include paths with the build process to generate code that remains portable and minimizes compatibility problems with future versions.

Assume that additional header files are:

```
c:\work\feature1\foo.h
c:\work\feature2\bar.h
```

- An approach is to include in the `#include` statements only the file name, such as:

  ```
  #include "foo.h"
  #include "bar.h"
  ```

  Then, the include path passed to the compiler contains folders in which the headers files exist:

  ```
  cc -Ic:\work\feature1 -Ic:\work\feature2     ...
  ```

- Another approach is to use relative paths in `#include` statements and provide an anchor folder for these relative paths using an include path, for example:

  ```
  #include "feature1\foo.h"
  #include "feature2\bar.h"
  ```

Then, specify the anchor folder (for example `\work`) to the compiler:

```
 cc -Ic:\work   ...
```

Avoid These Folder Dependencies

When using the build process, avoid dependencies on folders in the build process "Code generation folder" (Simulink), such as the *`model`*`_ert_rtw` folder or the `slprj` folder. Do not use paths in `#include` statements that are relative to the location of the generated source file. For example, if your MATLAB code generation folder is `c:\work`, the build process generates the *`model`*`.c` source file into a subfolder such as:

```
c:\work\model_ert_rtw\model.c
```

The *`model`*`.c` file has `#include` statements of the form:

```
#include "..\feature1\foo.h"
#include "..\feature2\bar.h"
```

It is preferable to use one of the other suggested approaches because the relative path creates a dependency on the code generator folder structure.

More About

- “Manage Build Process Folders” (Simulink Coder)
- “Manage Build Process Files” (Simulink Coder)
- “Manage Build Process File Dependencies” (Simulink Coder)
- “Enable Build Process for Folder Names with Spaces” (Simulink Coder)
- “Build and Run a Program” (Simulink Coder)

Enable Build Process for Folder Names with Spaces

The code generator uses alternate folder name support, which is specific to your operating system, to process folder names that include spaces. On Windows systems, the code generator maps a drive corresponding to the MATLAB installation folder for either of these conditions:

- The `matlabroot` folder is a UNC location.
- The path the `matlabroot` folder contains spaces, and the system has no alternate name support.

The build process provides similar support for other build-related folders. For a summary of support and limitations, see "Build Process Folder Support on Windows" (Simulink Coder).

Spaces in folder names can appear in the paths to build-related locations:

- `matlabroot`—path to your MATLAB installation folder.

  An example is a `matlabroot` similar to `C:\Program Files\MATLAB\R2015b`.
- `pwd`—current working folder from which you start the build.

  An example is a `pwd` at the start of a build similar to `C:\Users\username \Documents\My Work`.
- The installation folder for a compiler that the build process uses.

If your work environment includes one or more of the preceding scenarios, use the following support mechanisms for the build process:

- If you are using the toolchain approach to build generated code, the toolchain manages spaces in folder names by using alternate names from the platform (for example, Windows or UNIX). For more information about how the toolchain `TransformPathsWithSpaces` method manages these names, see addAttribute (coder.make.ToolchainInfo) (MATLAB Coder).
- If you are using the template makefile approach to build generated code, the template makefile (`.tmf`) requires code to manage spaces in folder names. When the alternate folder names (Windows short names) differ from the file system folder names (Windows long names), add this code to the makefile.

```
ALT_MATLAB_ROOT          = |>ALT_MATLAB_ROOT<|
ALT_MATLAB_BIN           = |>ALT_MATLAB_BIN<|
!if "$(MATLAB_ROOT)" != "$(ALT_MATLAB_ROOT)"
```

```
MATLAB_ROOT = $(ALT_MATLAB_ROOT)
!endif
!if "$(MATLAB_BIN)" != "$(ALT_MATLAB_BIN)"
MATLAB_BIN = $(ALT_MATLAB_BIN)
!endif
```

When the values of the location tokens are not equal, this code replaces `MATLAB_ROOT` with `ALT_MATLAB_ROOT`. The replacement indicates that the path to your MATLAB installation folder includes spaces. This code applies the same type of replacement for `MATLAB_BIN` with `ALT_MATLAB_BIN`. The preceding code is specific to `nmake`. For platform-specific examples, see the supplied template makefiles.

With either build approach, when there is an issue with support for creation of alternate names (short names), build errors can occur on Windows. If a build generates an error message similar to the following message, see "Troubleshooting Errors When Folder Names Have Spaces" (Simulink Coder).

```
NMAKE : fatal error U1073: don't know how to make ' ...
```

When using operating system commands, such as `system` or `dos`, enclose paths that specify executable files or command parameters in double quotes (`" "`). For example:

```
system('dir "D:\Applications\Common Files"')
```

Build Process Folder Support on Windows

Build Process Folders	Approach for Paths with UNC or Spaces	Support for Windows Platform

`matlabroot` folder
 	
Note: The `matlabroot` value is derived from the MATLAB installation location.
	During a build, a UNC location such as:
	
	`\\networkdrive\matlab\R20xxb`
	
	could be remapped as:
	
	`T:\`
	
	During a build on a Windows system with short file name (8.3) support (default for Windows using NTFS), the build process uses the Windows API `getShortPathName()` for the folder location.
		Build process folder support available independent of file system (NTFS or ReFS) or file system configuration for short file name support.
		
		Limitations:
		
		On systems that require drive mapping for the installation location, the build process requires that a drive letter is available for mapping.

Build Process Folders	Approach for Paths with UNC or Spaces	Support for Windows Platform

	During a build on a Windows system without short file name (8.3) support (systems using ReFS or using NTFS with 8.3 support disabled), a location with spaces in the path such as:
	`C:\Program Files\MATLAB\R20xxb`
	could be remapped as:
	`T:\R20xxb`
		On systems without short file name (8.3) support (using ReFS or using NTFS with 8.3 support disabled), the final folder in the installation location cannot contain spaces. For example, a final folder name:
		`C:\Program Files\MATLAB\R20xxb sp1`
		is not supported.

Build Process Folders	Approach for Paths with UNC or Spaces	Support for Windows Platform
Code generation folder Simulation cache folder Custom code source file locations—among others, these locations include folders specified by: • `rtwmakecfg.` • **Configuratio Parameters > Code Generation > Custom Code > Additional build information** • Code replacement library	For UNC locations, build process temporarily maps a drive by using the shell commands `pushd` and `popd`.	Build process folder support is available independent of file system (NTFS or ReFS) or file system configuration for short path name support.
	For paths with spaces, build process uses the Windows short path name (8.3) by using the Windows API: `getShortPathName()`	Build process folder support depends on NTFS file system and requires Windows default support. Registry sets value of 2 or 0 for: `NtfsDisable8dot3NameCreation` **Limitations:** Build process does not support spaces in the path to these folders for: • NTFS file system with short path name support disabled • ReFS file system (this file system does not support short path names)

Troubleshooting Errors When Folder Names Have Spaces

On Windows, when there is an issue with support for creation of short file names, build process errors can occur. When this issue affects a build, you see an error message similar to:

```
NMAKE : fatal error U1073: don't know how to make 'C:\Work\My'
```

This message can occur if a space in the folder name (`C:\Work\My Models`) prevents the build process from finding the model or a file to build. For descriptions of the build-

related folders that are sensitive to a space in the folder name or path, see "Build Process Folder Support on Windows" (Simulink Coder).

To avoid issues from folder names with spaces when Windows short file name support for file names is disabled, do not use paths with spaces. For example, install third-party software to paths without spaces. Do not use paths with spaces for folders containing your models, source files, or libraries.

An issue can occur with builds that use folder names with spaces, because it is possible to disable Windows alternate name support. The build process uses this alternate name support on Windows systems. There are many terms for this file, folder, and path alternate name support:

- 8.3 name
- DOS path
- short file name (SFN, ShortFileName)
- long name alias
- Windows path alias

Verify the type of file system that the drive uses. In Windows Explorer, right-click the drive icon and select properties.

- If the file system is ReFS (Resilient File System), it is an issue. The ReFS does not provide short file name support. Except for the MATLAB installation folder, the build process does not support folder names with spaces for the ReFS file system. If your work environment requires short file name support for the build folder or for additional external code folders, do not use ReFS.
- If the file system is NTFS (New Technology File System), it is possible that the build error is related to a registry setting incompatibility. Continue with troubleshooting steps.

The error could stem from an issue with short file name support on a system using NTFS. Check the Windows registry setting that enables the creation of short names for files, folders, and paths.

1. From the Windows Start menu, type `cmd`, right-click the `cmd.exe` icon, and select `Run as administrator`.
2. Change to the `windows\system32` folder and query the `NtfsDisable8dot3NameCreation` status by typing:

   ```
   > fsutil 8dot3name query
   ```

3. If the registry state of `NtfsDisable8dot3NameCreation` is not 2, the default (Volume level setting), change the value to 2 by typing:

   ```
   > fsutil 8dot3name set 2
   ```

   For more information about enabling creation of short names. See http://technet.microsoft.com/en-us/library/ff621566.aspx.

   Changing the registry setting enables creation of short names only for files and folders that are created after the change.

4. To create short names for files created while short name creation was disabled, at the Windows command line, use the `fsutil` utility.

   To set the short name, the syntax is:

   ```
   > fsutil file setshortname <FileName> <ShortName>
   ```

   For example, to create the short name `PROGRA~1` for the long name `C:\Program Files`, type:

   ```
   > fsutil file setshortname "C:\Program Files" PROGRA~1
   ```

   The `C:\Program Files` folder name is in quotations because it has spaces.

5. To verify that the short name was created, use the `dir` command with `/x` option to show short names.

   ```
   > dir C:\ /x
   ```

See Also

addAttribute (coder.make.ToolchainInfo) (MATLAB Coder)

More About

- "Manage Build Process Folders" (Simulink Coder)
- "Manage Build Process Files" (Simulink Coder)
- "Manage Build Process File Dependencies" (Simulink Coder)
- "Add Build Process Dependencies" (Simulink Coder)
- "Build and Run a Program" (Simulink Coder)

External Websites

- MATLAB Answers: "Why is the build process failing ...?"

- http://technet.microsoft.com/en-us/library/cc788058.aspx
- http://technet.microsoft.com/en-us/library/cc788058.aspx

Code Generation of Matrices and Arrays

In this section...

MATLAB, Simulink, and the code generator store matrix data and arrays (1–D, 2–D, ...) in column-major format as a vector. Column-major format orders elements in a matrix starting from the first column, top to bottom, and then moving on to the next column. For example, the following 3x3 matrix:

```
A =
     1   2   3
     4   5   6
     7   8   9
```

translates to an array of length 9 in the following order:

```
A(1) = A(1,1) = 1;
A(2) = A(2,1) = 4;
A(3) = A(3,1) = 7;
A(4) = A(1,2) = 2;
A(5) = A(2,2) = 5;
```

and so on.

In column-major format, the next element of an array in memory is accessed by incrementing the first index of the array. For example, these element pairs are stored sequentially in memory:

- `A(i)` and `A(i+1)`
- `B(i,j)` and `B(i+1,j)`
- `C(i,j,k)` and `C(i+1,j,k)`

For more information on the internal representation of MATLAB data, see “MATLAB Data” (MATLAB) in the MATLAB External Interfaces document.

Code generation software uses column-major format for several reasons:

- The world of signal and array processing is largely column major: MATLAB, LAPack, Fortran90, DSP libraries.

- A column is equivalent to a channel in frame-based processing. In this case, column-major storage is more efficient.
- A column-major array is self-consistent with its component submatrices:
  - A column-major 2–D array is a simple concatenation of 1–D arrays.
  - A column-major 3–D array is a simple concatenation of 2–D arrays.
  - The stride is the number of memory locations to index to the next element in the same dimension. The stride of the first dimension is one element. The stride of the nth dimension element is the product of sizes of the lower dimensions.
  - Row-major n-D arrays have their stride of 1 for the highest dimension. Submatrix manipulations are typically accessing a scattered data set in memory, which does not allow for efficient indexing.

C typically uses row-major format. MATLAB and Simulink use column-major format. You cannot configure the code generation software to generate code with row-major ordering. If you are integrating external C code with the generated code, consider the following:

To	Consider
Integrate row-major data in external C code with Simulink generated functions.	• Transposing the row-major data in your external C code into column-major format as a 1–D array. • Using "Access Data Through Functions with Custom Storage Class `GetSet`" on page 23-92. The `GetSet` custom storage class replaces get (read from) and set (write to) with user-specified `GetSet` functions. The user-specified `GetSet` function has a single index argument, requiring the function to process the argument and access the targeted row and column data element. Using the `GetSet` function impacts code efficiency.
Integrate external C code functions requiring row-major data with Simulink generated data.	Using the Legacy Code Tool option `convert2DMatrixToRowMajor` to create an S-function that integrates external code functions with Simulink generated data. See `legacy_code`. The generated code stores the data as a 1–D array. Using `convert2DMatrixToRowMajor` impacts code efficiency.

Code Generator Matrix Parameters

The compiled model file, *model*.rtw, represents matrices as character vectors in MATLAB syntax, without an implied storage format. This format allows you to copy the character vector out of an .rtw file, paste it into a MATLAB file, and have it recognized by MATLAB.

The code generator declares Simulink block matrix parameters as scalar or 1-D array variables

```
real_T scalar;
real_T mat[ nRows * nCols ];
```

real_T can be a data type supported by Simulink. It matches the variable type given in the model file.

For example, the 3-by-3 matrix in the 2–D Look-Up Table block

```
    1   2   3
    4   5   6
    7   8   9
```

is stored in *model*.rtw as

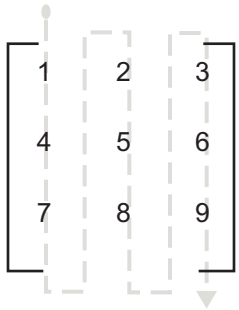
```
Parameter {
  Name   "OutputValues"
  Value   Matrix(3,3)
[[1.0, 2.0, 3.0]; [4.0, 5.0, 6.0]; [7.0, 8.0, 9.0];]
  String  "t"
  StringType   "Variable"
  ASTNode {
     IsNonTerminal  0
     Op  SL_NOT_INLINED
     ModelParameterIdx 3
    }
}
```

and results in this definition in *model*.h.

```
typedef struct Parameters_tag {
  real_T s1_Look_Up_Table_2_D_Table[9];
  /* Variable:s1_Look_Up_Table_2_D_Table
   * External Mode Tunable:yes
   * Referenced by block:
```

```
   * <S1>/Look-Up Table (2-D
   */

  [ ... other parameter definitions ... ]

} Parameters;
```

The *model*.h file declares the actual storage for the matrix parameter. You can see that the format is column-major.




```
Parameters model_P = {
  /* 3 x 3 matrix s1_Look_Up_Table_2_D_Table */
  { 1.0, 4.0, 7.0, 2.0, 5.0, 8.0, 3.0, 6.0, 9.0 },
  [ ... other parameter declarations ...]
};
```

Internal Data Storage for Complex Number Arrays

Simulink and code generator internal data storage formatting differs from MATLAB internal data storage formatting only in the storage of complex number arrays. In MATLAB, the real and imaginary parts are stored in separate arrays. In Simulink and the code generator, the parts are stored in an "interleaved" format. The numbers in memory alternate real, imaginary, real, imaginary, and so forth. This convention allows efficient implementations of small signals on Simulink lines, for Mux blocks, and other "virtual" signal manipulation blocks. For example, the signals do not actively copy their inputs, just the references.

Generate Shared Utility Code

Blocks in a model can require common functionality to implement their algorithm. Consider modularizing this functionality into standalone support or helper functions. This approach can be more efficient than inlining the code for the functionality for each block instance. These points help with the decision to use a library or a shared utility:

- Package functions that can have multiple callers into a library when the functions are defined statically. That is, the function source code exists in a file before you use the code generator to produce code for your model.
- Package functions that can have multiple callers as shared utilities (produced during code generation) when the functions cannot be defined statically. For example, several model- and block-specific properties specify which functions are used and their behavior. Also, these properties determine type definitions (for example, `typedef`) in shared utility header files. The number of possible combinations of properties that determine unique behavior make it impractical to define statically the possible function files before code generation.

In this section...

Control Placement of Shared Utility Code

Control placement of shared utility code with the **Shared code placement** option in **Configuration Parameters** > **Code Generation** > **Interface** pane. The default option value is `Auto`. For this setting, the code generator places code required for fixed-point and other utilities in the *`model`*`.c` file, the *`model`*`.cpp` file, or a separate file in the build folder (for example, `vdp_grt_rtw`) if a model does not contain existing shared utility code or one of the following blocks:

- Model blocks
- Simulink Function blocks
- Function Caller blocks

- calls to Simulink Functions from Stateflow or MATLAB Function blocks

If a model contains one or more of the above blocks, the code generator creates and uses a shared utilities folder within `slprj`. The naming convention for shared utility folders is `slprj/target/_sharedutils`. *`target`* is `sim` for simulations with Model blocks or the name of the system target file for code generation.

```
slprj/sim/_sharedutils      % folder used with simulation
slprj/grt/_sharedutils      % folder used with grt.tlc STF
slprj/ert/_sharedutils      % folder used with ert.tlc STF
slprj/mytarget/_sharedutils % folder used with mytarget.tlc STF
```

To force a model build to use the `slprj` folder for shared utility generation, even when the current model does not contain existing shared utility code or one of the blocks listed above, set the **Shared code placement** option to `Shared location`. The code generator places utilities under the `slprj` folder rather than in the normal build folder. This setting is useful when you are manually combining code from several models, as it prevents symbol collisions between the models.

Control Placement of rtwtypes.h for Shared Utility Code

The generated `rtwtypes.h` header file provides fundamental type definitions, `#define` statements, and enumerations. For more information, see “rtwtypes.h” on page 33-49.

Control placement of rtwtypes.h file by selecting whether the build process uses the shared utilities folder. If the model build uses a shared utilities folder, the build process places `rtwtypes.h` in `slprj/target/_sharedutils`. Otherwise, the software places `rtwtypes.h` in *`model_target`*`_rtw`.

Adding a model to a model hierarchy or changing an existing model in the hierarchy can result in updates to the shared `rtwtypes.h` file during code generation. If updates occur, recompile and, depending on your development process, reverify previously generated code. To minimize updates to the `rtwtypes.h` file, make the following changes in the Configuration Parameters dialog box:

- On the **Interface** pane, select **Support: complex numbers** even if the model does not currently use complex data types. Selecting this option protects against a future requirement to add support for complex data types when integrating code.
- On the **All Parameters** tab, clear **Support: non-inlined S-functions**. If you use noninlined S-functions in the model, this option generates an error.
- On the **All Parameters** tab, clear **Classic call interface**. Disables use of the GRT interface.

Avoid Duplicate Header Files for Exported Data

The exported header files appear in the shared utility folder when:

- You control the file placement of declarations for signals, parameters, and states by applying storage classes and custom storage classes.
- The code generator places utility code in a shared location.

For example, you can specify a header file for a piece of data through:

- The **Code Generation** tab in a Signal Properties dialog box.
- The `HeaderFile` property of a data object. Data objects are objects of the classes `Simulink.Signal` and `Simulink.Parameter`.

If you want the declaration to appear in the file *`model`*`.h`, it is a best practice to leave the header file name unspecified. By default, the code generator places data declarations in *`model`*`.h`.

If you specify *`model`*`.h` as the header file name, and if the code generator places utility code in a shared location, you cannot generate code from the model. The code generator cannot create the file *`model`*`.h` in both the model build folder and the shared utility folder.

Reduce Shared Utility Code Generation with Incremental Builds

You can specify that the model build generates C source files in a shared utilities folder. See “Control Placement of Shared Utility Code” on page 33-80. These files include C source files that contain function definitions and header files that contain macro definitions. For this discussion, the term functions means functions and macros.

Blocks within the same model and blocks in different models can use a shared function when you use model reference or when you build multiple standalone models from the same start build folder. The code generator produces the code for a given function only once for the block that first triggers code generation. As the product determines the requirement to generate function code for subsequent blocks, it performs a file existence check. If the file exists, the model build does not regenerate the function. The shared utility code mechanism requires that a given function and file name represent the same functional behavior regardless of which block or model generates the function. To satisfy this requirement:

- Model properties that determine function behavior contribute to the shared utility checksum or determine the function and file name.
- Block properties that determine the function behavior also determine the function and file name.

During compilation, makefile rules for the shared utilities folder select compilation of only new C files and incrementally archive the object file into the shared utility library, `rtwshared.lib`, or `rtwshared.a`. Incremental compilation also occurs.

More About

- "Manage the Shared Utility Code Checksum" (Simulink Coder)
- "Generate Shared Utility Code for Fixed-Point Functions" (Simulink Coder)
- "Generate Shared Utility Code for Custom Data Types" (Simulink Coder)
- "Cross-Release Shared Utility Code Reuse" (Simulink Coder)

Manage the Shared Utility Code Checksum

When a model configuration sets **Configuration Parameters** > **Code Generation** > **Interface** > **Shared code placement** as **Shared location** or when the model contains Model blocks, the code generator places the shared code in the shared utilities folder. The build process generates a shared utilities checksum of the code generation configuration for the shared code.

During subsequent code generation, if the checksum file `slprj/target/_sharedutils/checksummap.mat` exists relative to the current folder, the code generator reads that file. The code generator verifies that the current model that you are building has configuration properties that match the checksum of properties from the shared utility model. If mismatches occur between the properties defined in `checksummap.mat` and the current model properties, you see an error. Use the error message to manage the checksum (for example, diagnose and resolve the configuration issues with the current model).

For more information, see "Reduce Shared Utility Code Generation with Incremental Builds" on page 33-82.

In this section...

View the Shared Utility Checksum Hash Table

It is helpful to view the property values that contribute to the checksum. This example uses the `rtwdemo_lct_start_term.slx` model. To load the `checksum.mat` file into MATLAB and view the `targetInfoStruct` that defines the checksum-related properties:

1. Open the `rtwdemo_lct_start_term.slx` model. In the Command Window, type:

   ```
   rtwdemo_lct_start_term
   ```

2. Create and move to a new working folder.

   ```
   mkdir C:\Temp\demo
   cd C:\Temp\demo
   ```

3. Save a copy of the model in the folder.
4. Build the model by using the **Simulink** > **Code** > **C/C++ Code** > **Build Model** command. This model is already set up to produce shared utilities.

5. Move to the `_sharedutils` folder created by the build process.

```
cd C:\Temp\demo\slprj\grt\_sharedutils
```

6. Load the `checksummap.mat` file into MATLAB.

```
load checksummap
```

7. Display the contents of `hashTbl.targetInfoStruct` and examine the checksum-related property values.

```
hashTbl.targetInfoStruct
```

For this example, the Command Window displays `hashTbl.targetInfoStruct` for the shared utilities that you generated from the model:

```
                ShiftRightIntArith: 'on'
            ProdShiftRightIntArith: 'on'
                        Endianess: 'LittleEndian'
                    ProdEndianess: 'LittleEndian'
                      wordlengths: '8,16,32,32,64,32,64,64,64,64'
                  Prodwordlengths: '8,16,32,32,64,32,64,64,64,64'
                   TargetWordSize: '64'
                     ProdWordSize: '64'
                TargetHWDeviceType: 'Custom Processor->MATLAB Host Computer'
                  ProdHWDeviceType: 'Intel->x86-64 (Windows64)'
               TargetIntDivRoundTo: 'Zero'
                 ProdIntDivRoundTo: 'Zero'
                           tmfName: ''
                      toolchainName: ''
                           computer: 'PCWIN64'
UseDivisionForNetSlopeComputation: 'off'
                PurelyIntegerCode: 'off'
                PortableWordSizes: 'off'
            SupportNonInlinedSFcns: ''
                   TargetLibSuffix: ''
               RTWReplacementTypes: ''
                         MaxIdInt8: 'MAX_int8_T'
                         MinIdInt8: 'MIN_int8_T'
                        MaxIdUint8: 'MAX_uint8_T'
                        MaxIdInt16: 'MAX_int16_T'
                        MinIdInt16: 'MIN_int16_T'
                       MaxIdUint16: 'MAX_uint16_T'
                        MaxIdInt32: 'MAX_int32_T'
                        MinIdInt32: 'MIN_int32_T'
                       MaxIdUint32: 'MAX_uint32_T'
                    BooleanTrueId: 'true'
                   BooleanFalseId: 'false'
 TypeLimitIdReplacementHeaderFile: ''
             SharedCodeRepository: ''
                      TargetLang: 'C'
        PreserveExternInFcnDecls: 'on'
         EnableSignedRightShifts: 'on'
          EnableSignedLeftShifts: 'on'
                         TflName: 'None'
```

```
                    TflCheckSum: [1.6615e+09 521991164 2.2147e+09 1.7704e+09]
                 UtilMemSecName: 'Default'
           CodeCoverageChecksum: [3.6498e+09 78774415 2.5508e+09 2.1183e+09]
     TargetLargestAtomicInteger: 'Char'
       TargetLargestAtomicFloat: 'None'
       ProdLargestAtomicInteger: 'Char'
         ProdLargestAtomicFloat: 'Float'
                   LongLongMode: 'on'
               ProdLongLongMode: 'off'
CollapseNonTrivialExpressions: 'false'
```

Relate the Shared Utility Checksum to Configuration Parameters

Examine the `targetInfoStruct` hash table from the shared utility model. Some key-value pairs relate directly to a model property. Other pairs relate to groups of properties.

The following table describes the key-value pairs.

Key Names	Model Properties
ShiftRightIntArith	`TargetShiftRightIntArith`
ProdShiftRightIntArith	`ProdShiftRightIntArith`
Endianess	`TargetEndianess`
ProdEndianess	`ProdEndianess`
wordlengths	`TargetBitPerChar`, `TargetBitPerShort`, `TargetBitPerInt`, `TargetBitPerLong`, `TargetBitPerLongLong`, `TargetBitPerFloat`, `TargetBitPerDouble`, `TargetBitPerPointer`
Prodwordlengths	`ProdBitPerChar`, `ProdBitPerShort`, `ProdBitPerInt`, `ProdBitPerLong`, `ProdBitPerLongLong`, `ProdBitPerFloat`, `ProdBitPerDouble`, `ProdBitPerPointer`
TargetWordSize	`TargetWordSize`
ProdWordSize	`ProdWordSize`
TargetHWDeviceType	`TargetHWDeviceType`
ProdHWDeviceType	`ProdHWDeviceType`

Key Names	Model Properties
TargetIntDivRoundTo	`TargetIntDivRoundTo`
ProdIntDivRoundTo	`ProdIntDivRoundTo`
tmfName	`TemplateMakefile`
toolchainName	`Toolchain`
computer	return value of the `computer` command
UseDivisionForNetSlopeComputation	`UseDivisionForNetSlopeComputation`
PurelyIntegerCode	`PurelyIntegerCode`
PortableWordSizes	`PortableWordSizes`
SupportNonInlinedSFcns	`SupportNonInlinedSFcns`
TargetLibSuffix	Removed (not used)
RTWReplacementTypes	`EnableUserReplacementTypes`, `ReplacementTypes`
MaxIdInt8	`MaxIdInt8`
MinIdInt8	`MinIdInt8`
MaxIdUint8	`MaxIdUint8`
MaxIdInt16	`MaxIdInt16`
MinIdInt16	`MinIdInt16`
MaxIdUint16	`MaxIdUint16`
MaxIdInt32	`MaxIdInt32`
MinIdInt32	`MinIdInt32`
MaxIdUint32	`MaxIdUint32`
BooleanTrueId	`BooleanTrueId`
BooleanFalseId	`BooleanFalseId`
TypeLimitIdReplacementHeaderFile	`TypeLimitIdReplacementHeaderFile`
SharedCodeRepository	reserved (internal use only)
TargetLang	`TargetLang`
PreserveExternInFcnDecls	`PreserveExternInFcnDecls`
EnableSignedRightShifts	`EnableSignedRightShifts`

Key Names	Model Properties
EnableSignedLeftShifts	`EnableSignedLeftShifts`
TflName	`CodeReplacementLibrary`
TflCheckSum	reserved (internal use only)
UtilMemSecName	`MemSecFuncSharedUtil`, `MemSecPackage`
CodeCoverageChecksum	reserved (internal use only)
TargetLargestAtomicInteger	`TargetLargestAtomicInteger`
TargetLargestAtomicFloat	`TargetLargestAtomicFloat`
ProdLargestAtomicInteger	`ProdLargestAtomicInteger`
ProdLargestAtomicFloat	`ProdLargestAtomicFloat`
LongLongMode	`TargetLongLongMode`
ProdLongLongMode	`ProdLongLongMode`
CollapseNonTrivialExpressions	reserved (internal use only)

More About

- “Generate Shared Utility Code” (Simulink Coder)
- “Generate Shared Utility Code for Fixed-Point Functions” (Simulink Coder)
- “Generate Shared Utility Code for Custom Data Types” (Simulink Coder)
- “Cross-Release Shared Utility Code Reuse” (Simulink Coder)

Generate Shared Utility Code for Fixed-Point Functions

An important set of generated functions that the model build places in the shared utility folder are the fixed-point support functions. Based on model and block properties, there are many possible versions of fixed-point utilities functions that make it impractical to provide a complete set as static files. Generating only the required fixed-point utility functions during the code generation process is an efficient alternative.

The shared utility checksum mechanism makes sure that several critical properties are identical for models that use the shared utilities. For the fixed-point functions, there are additional properties that determine function behavior. The mechanism codes these properties into the functions and file names to maintain requirements. The additional properties include:

Category	Function/Property
Block properties	• Fixed-point operation that the block performs • Fixed-point data type and scaling (`Slope`, `Bias`) of function inputs and outputs • Overflow handling mode (`Saturation`, `Wrap`) • Rounding Mode (`Floor`, `Ceil`, `Zero`)
Model properties	`get_param(bdroot, 'NoFixptDivByZeroProtection')`

The property-based naming convention for the fixed-point utilities is as follows:

```
operation + [zero protection] + output data type + output bits +
[input1 data] + input1 bits + [input2 data type + input2 bits] +
[shift direction] + [saturate mode] + [round mode]
```

The file names shown are examples of generated fixed-point utility files. The function or macro names in the file are identical to the file name without the extension.

```
FIX2FIX_U12_U16.c
FIX2FIX_S9_S9_SR99.c
ACCUM_POS_S30_S30.h
MUL_S30_S30_S16.h
div_nzp_s16s32_floor.c
div_s32_sat_floor.c
```

For these examples, the table shows how the respective fields correspond.

The `ACCUM_POS` example uses the output variable as one of the input variables. So, the file and macro name only contain the output and second input.

The second `div` example has identical data type and bits for both inputs and the output. So, the file and function name only include the output.

Operation	FIX2FIX	FIX2FIX	ACCUM_POS	MUL	div	div
Zero protection	NULL	NULL	NULL	NULL	_nzp	NULL
Output data type	_U	_S	_S	_S	_s	_s
Output bits	12	9	30	30	16	32
Input data type	_U	_S	_S	_S [and _S]	s	NULL
Input bits	16	9	30	30 [and 16]	32	NULL
Shift direction	NULL	SR99	NULL	NULL	NULL	NULL
Saturate mode	NULL	NULL	NULL	NULL	NULL	_sat
Round mode	NULL	NULL	NULL	NULL	_floor	_floor

More About

- “Generate Shared Utility Code” (Simulink Coder)
- “Manage the Shared Utility Code Checksum” (Simulink Coder)
- “Generate Shared Utility Code for Custom Data Types” (Simulink Coder)
- “Cross-Release Shared Utility Code Reuse” (Simulink Coder)
- “Control the Generation of Fixed-Point Utility Functions” (Fixed-Point Designer)

Generate Shared Utility Code for Custom Data Types

By default, if a model employs a custom data type (such as a `Simulink.AliasType` object or an enumeration class), the code generator places the corresponding type definition (`typedef`) in the *`model`*`_types.h` file. When you generate code from multiple models, each model duplicates the type definition. These duplicate definitions can prevent you from compiling the bodies of generated code together.

However, you can configure the code generator to place a single type definition in a header file in the `_sharedutils` folder. Then, when you generate code from a model, if the type definition already exists in the `_sharedutils` folder, the code generator does not duplicate the definition, but instead reuses it through inclusion (`#include`).

Through this mechanism, you can share:

- Simulink data type objects that you instantiate from the classes `Simulink.AliasType`, `Simulink.Bus`, and `Simulink.NumericType`. For basic information about creating and using these objects, see “What Are User-Defined Data Types?” on page 21-2 and Simulink.Bus.
- Enumerations that you define, for example, by authoring an `enum` class in a script file or by using the function `Simulink.defineIntEnumType`. For basic information about defining enumerations in Simulink, see “Use Enumerated Data in Simulink Models” (Simulink).

To share a custom data type across multiple models:

1. Define the data type. For example, create the `Simulink.AliasType` object.
2. Set data scope and header file properties to specific values that enable sharing.

   For a data type object, set the `DataScope` property to `'Exported'` and, optionally, specify the header file name through the `HeaderFile` property.

   For an enumeration that you define as an `enum` class in a script file, implement the `getDataScope` method (with return value `'Exported'`) and, optionally, implement the `getHeaderFile` method.

   For an enumeration that you define by using the `Simulink.defineIntEnumType` function, set the `'DataScope'` pair argument to `'Exported'` and, optionally, specify the `'HeaderFile'` pair argument
3. Use the data type in the models.

4. Before generating code from each model, set **Configuration Parameters** > **Code Generation** > **Interface** > **Shared code placement** to `Shared location`.
5. Generate code from the models.

Note: You can configure the definition of the custom data type to appear in a header file in the `_sharedutils` folder. The shared utility functions that the model build generates into the `_sharedutils` folder do not use the custom data type name. Only model code located in code folders for each model uses the custom data type name.

More About

- "Generate Shared Utility Code" (Simulink Coder)
- "Manage the Shared Utility Code Checksum" (Simulink Coder)
- "Generate Shared Utility Code for Fixed-Point Functions" (Simulink Coder)
- "Cross-Release Shared Utility Code Reuse" (Simulink Coder)
- "Control File Placement of User-Defined Types" on page 21-6

Cross-Release Shared Utility Code Reuse

In this section...

When you generate code for a model, the code generator by default creates shared utility files that the model requires. When you generate code with different releases, the code generators can produce functionally identical shared files that contain some nonfunctional differences. For example, different comments and different coding style. When you use the same release to generate code for different models in different folders, you can also produce shared files with nonfunctional differences. For example, if you specify different `ParenthesesLevel` or `ExpressionFolding` values for the models, the code generator can produce shared files that contain different comments or different coding styles.

Integrated code that includes functionally identical shared files:

- Is more expensive to verify because each shared file requires verification.
- Produces compilation errors if the shared files define duplicate symbols.

If you have an Embedded Coder license, you can avoid these issues by specifying the reuse of shared code from an existing folder, for example, a read-only library of verified code. In this case, the code generator does not create new shared utility files. The build process uses external code or previously generated shared utility code from the folder. An administrator maintains and updates the read-only library.

Workflow to Reuse Shared Utility Code

1. In the **Configuration Parameters > All Parameters > Existing shared code** field, enter the full path to your shared code folder.
2. Verify that the **Configuration Parameters > All Parameters > Use only existing shared code** diagnostic is set to `error` (default).
3. Remove the `slprj` folder or move to a new working folder.
4. Build your model. If you do not see an error, your shared code folder contains the required shared utility files.
5. If files are missing from the shared code folder, you see an error. To continue code generation with a locally generated version of the missing shared utility files:

a Set **Configuration Parameters > All Parameters > Use only existing shared code** to `warning`.

b Rebuild your model. The code generation process uses a locally generated version of the missing shared utility files.

c Provide the administrator of the verified code library with your model and information about missing shared utility files. With the model, the administrator generates the required shared utility files. Using `sharedCodeUpdate`, the administrator adds the files to the shared code folder.

d When the files are available in the shared code folder, repeat steps 1–4.

If the shared utility code is generated from library subsystems that are shared across models (Simulink Coder), you cannot reuse the code *across* releases because the code is release-specific—the symbol name and file name mangling includes the release number. The administrator must add the shared utility code generated for each release to the shared code folder.

The `sharedCodeUpdate` function can add files to the shared code folder that have identical content but different file and function names. This behavior is useful when you have different model components that require their own shared utility functions. Although some code is duplicated, the different model components can access the shared utility functions with which they were verified. To force model components to have their own versions of shared utility functions, configure naming rules to insert the model name into shared utility identifiers (Simulink Coder).

Required Edits to Reuse Shared Utility Code

For most shared utility code files, you can specify master copies that you can reuse across releases without modifying the files. With some files, for example, `rtwtypes.h`, and `zero_crossing_types.h`, there are situations where manual editing is required to produce master copies that you can use with generated code from different releases. For example:

- The `rtwtypes.h` file generated by R2010a contains a checksum.

  ```
  /* This ID is used to detect inclusion of an incompatible rtwtypes.h */
  #define RTWTYPES_ID_C08S16I32L64N64F0
  ```

  For each R2010a version of `rtwtypes.h` that you want to include in your integration, copy the corresponding `#define` statement into your master copy of `rtwtypes.h`.

- In R2015a, the zero-crossing definitions moved from `rtwtypes.h` into `zero_crossing_types.h`. To create an `rtwtypes.h` file that is compatible with generated model code from different releases, in your master copy of `rtwtypes.h`, insert this statement.

  ```
  #include "zero_crossing_types.h"
  ```
  Remove definitions from `rtwtypes.h` that `zero_crossing_types.h` provides.

See Also

`crossReleaseImport` | `crossReleaseExport` | `sharedCodeUpdate`

Related Examples

- “Cross-Release Code Integration” on page 33-96

More About

- “Generate Shared Utility Code” (Simulink Coder)
- “Manage the Shared Utility Code Checksum” (Simulink Coder)
- “Generate Shared Utility Code for Fixed-Point Functions” (Simulink Coder)
- “Generate Shared Utility Code for Custom Data Types” (Simulink Coder)

Cross-Release Code Integration

In this section...

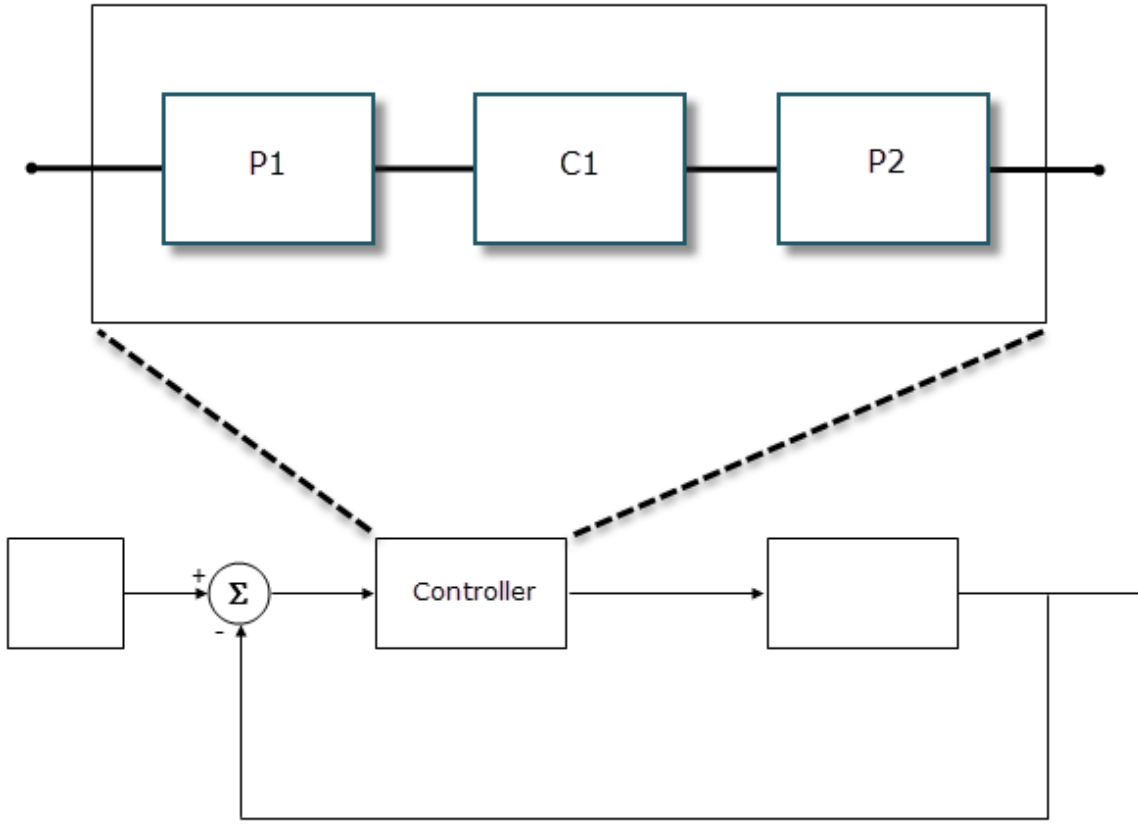
If you have an Embedded Coder license, you can integrate generated C code from previous releases (R2010a and later) with generated code from the current release when:

- The source models are single-rate.
- The source models are set to generate nonreusable code with function prototype control (root-level Inport and Outport blocks are mapped to step function arguments).
- The generated C code is from top-model and subsystem build processes.

If you can reuse existing code without modification, you can reduce the cost of reverification.

Workflow

Consider this control system model.



The `Controller` Model block references a model that consists of three components:

- `P1` is a Model block, which references a model developed with a previous release, for example, R2015b. The generated model code, with the standalone code interface, is in the folder `P1_ert_rtw`.
- `C1` is a Model block, which references a model that you are developing in the current release.
- `P2` is a subsystem block developed with a previous release, for example, R2016a. The generated subsystem code is in the folder `P2_ert_rtw`.

To integrate code from previous releases with generated code from the current release, use this workflow:

1 Export components from previous releases

a Add the `crossReleaseExport` function to the search paths for previous releases. For example, in the Command Window of a previous release, run this command:

```
addpath(fullfile(matlabRootCR, 'toolbox','coder','xrelexport'));
```

`matlabRootCR` is the `matlabroot` value for your current release.

b Using the previous releases, export generated component code. For example:

- From R2015b, run:

  ```
  crossReleaseExport(fullfile(folderPath,'P1_ert_rtw'))
  ```

- From R2016a, run:

  ```
  crossReleaseExport(fullfile(folderPath,'P2_ert_rtw'))
  ```

The `crossReleaseExport` function creates export artifacts in new folders `P1_R2015b` and `P2_R2016a`.

2 Specify an existing shared code folder

You can specify the reuse of shared code from an existing folder, for example, a library of verified code that an administrator maintains and updates. Specify the shared code folder for model components that require code generation in the current release, for example, `Controller` and `C1`.

In the **Configuration Parameters > All Parameters > Existing shared code** field, enter the full path to the shared code folder.

3 Import components into current release

From the current release, using export artifacts created in step 2, import generated component code from previous releases as software-in-the-loop (SIL) blocks or processor-in-the-loop (PIL) blocks. For example:

```
crossReleaseImport(P1_R2015bFullPath,'Controller', ...
'SimulationMode','SIL');
crossReleaseImport(P2_R2016aFullPath,'Controller', ...
'SimulationMode','SIL');
```

`P1_R2015bFullPath` and `P2_R2016aFullPath` are paths to the export artifact folders, `P1_R2015b` and `P2_R2016a`, created in step 1.

The `crossReleaseImport` function creates software-in-the-loop (SIL) blocks and subfolders:

- `P1_R2015b_sil` and `P1_R2015b_sil_resources`
- `P2_R2016a_sil` and `P2_R2016a_sil_resources`

4 Incorporate components into current release model

To replace components with SIL or PIL blocks, use the Simulink Editor or the `pil_block_replace` command. For example:

- Replace `P1` with `P1_R2015b_sil`.
- Replace `P2` with `P1_R2016a_sil`.

When you run a model simulation, the simulation runs the previous release code through the SIL or PIL blocks.

When you build the `Controller` model (`rtwbuild('Controller')`), the code generator does not generate new code for the components represented by the SIL or PIL blocks. The model code calls code generated by previous releases.

Limitations

The cross-release code integration workflow does not support:

- Export-function models.
- Model blocks inside the exported component. For a workaround, see “Incorporate Model Reference Code” on page 33-100.
- AUTOSAR code generation.
- Simulink Function, Function Caller, and Data Store Memory blocks across the boundaries of code generated by different releases.
- Parameter tuning through the SIL or PIL blocks.
- The integration of generated code from releases before R2010a.
- The import of generated code from the current release into a previous release (forward compatibility).
- The export of files located in the MATLAB root folder of the previous release, for example, blockset library files.
- The export and import of generated code from models with non-inlined S-functions.

- C-API on page 43-2.

You can export a Model block component only if the referenced model has these settings:

- `ModelReferenceNumInstancesAllowed` is `Single`.
- `SuppressErrorStatus` is `on`.

At the end of the model build process, the code generation report displays shared files that are directly used by the integration model, for example, `Controller`. The report does not display shared files used by the components of the model, for example, `P1` and `P2`.

Incorporate Model Reference Code

Suppose a Model block in a component references a model through the model reference code interface. When you use `crossReleaseExport` to export generated code from a previous release, the function does not export the referenced model code. You see a warning:

```
Cross-release export does not support model references. Generated code
for the following models will not be exported:
```

To work around this limitation, before running `crossReleaseImport`, use the `sharedCodeUpdate` function to import the generated code for the referenced model into your specified shared code folder.

```
sharedCodeUpdate(referencedModelCodeFolder,existingSharedCodeFolder);
```

The function creates a subfolder for the copied referenced model code.

If you try to import the component code without the referenced model code, `crossReleaseImport` produces a build error.

`Simulink.Bus` Support

To use a bus object as a data type in cross-release code integration, use one of these approaches.

Approach	Details
Exported bus	In the previous release, before generating code, specify these properties of the Simulink.Bus object: • `DataScope` — Set to `Exported`.

Approach	Details
	• `HeaderFile` — Specify a file name, for example, `prevRelBusType`. The code generator creates `prevRelBusType.h` in the shared utility code folder. This header file contains the definition for the `Simulink.Bus` data type. Use `sharedCodeUpdate` to add `prevRelBusType.h` to the shared code folder that `ExistingSharedCode` specifies. For R2010a and R2010b, the `DataScope` property is not available. Do not assign a value to the `HeaderFile` property. The code generator creates the `Simulink.Bus` data type definition in `modelName_types.h`, which is located in the code generation folder for the model.
	In the current release, before running `crossReleaseImport`, set the `DataScope` property of the Simulink.Bus object to `Imported`. When you build the integration model that incorporates the imported SIL or PIL block, the build process uses the `Simulink.Bus` data type definition in `prevRelBusType.h`. If the imported code is from R2010a or R2010b, specify these properties of the Simulink.Bus object: • `DataScope` — Set to `Imported`. • `HeaderFile` — Set to file path for `modelName_types.h`, which is in the imported code folder. When you build the integration model, the build process uses the `Simulink.Bus` data type definition in `modelName_types.h`.
Imported bus	In the previous release, before generating code, specify these properties of the Simulink.Bus object: • `DataScope` — Set to `Imported`. • `HeaderFile` — Specify a path to a file that contains the `Simulink.Bus` data type definition, for example, `aBusType.h`. For R2010a and R2010b, the `DataScope` property is not available. For the `HeaderFile` property, specify a path to a file that contains the `Simulink.Bus` data type definition, for example, `aBusType.h`.

Approach	Details
	In the current release, after importing generated code, you do not have to change the Simulink.Bus. When you build the integration model that incorporates the imported SIL or PIL block, the build process uses the `Simulink.Bus` data type definition from `aBusType.h`. If the imported code is from R2010a or R2010b, specify these properties of the Simulink.Bus object: • `DataScope` — Set to `Imported`. • `HeaderFile` — Set to `aBusType.h`. When you build the integration model, the build process uses the `Simulink.Bus` data type definition in `aBusType.h`.

Parameter Tuning

For a component from a previous release, you can export a block parameter `P` as an inline value in generated code. Before generating code:

- For R2015a and earlier releases, set `InlineParameters` or `InlineParams` to `on`.
- For R2015b and later releases, set `DefaultParameterBehavior` to `Inlined`.

To control tunability in the integration model through a single instance of `P`:

1. Before generating code in the previous release, apply the `ExportedGlobal` storage class to the `Simulink.Parameter` object for `P`.
2. In the integration model, use the `ImportedExtern` storage class to reference `P`.

You can use the `SimulinkGlobal` storage class for the exported component and integration model. This approach produces separate parameter definitions and values for `P`: one set for the component from the previous release and another set for the rest of the integration model.

On a Macintosh OS X system, which supports the Clang compiler, a top-model SIL or PIL simulation of the integration model with default **Build configuration** settings produces a compilation error. To work around this limitation, modify the default settings:

1. Get the build tool from the default toolchain.

```
tc = coder.make.getDefaultToolchain;
cComp = tc.getBuildTool('C Compiler');
```

2. Extract the C compiler standard options.

```
stdMaps = cComp.SupportedStandard.getLangStandardMaps;
optionValues = stdMaps.getCompilerOptions('*');
```

3. Remove `-fno-common` from the standard options for the C and C++ compilers.

```
optionToRemove = '-fno-common';
optionsToKeep = strrep(optionValues, optionToRemove, '');
c_standard_opts_id = '$(C_STANDARD_OPTS)';

custToolChainOpts = get_param(model,'CustomToolchainOptions');
custToolChainOpts{2} = strrep(custToolChainOpts{2}, c_standard_opts_id, optionsToKeep);

set_param(model, 'CustomToolchainOptions',custToolChainOpts);
```

Compare Simulation Behavior of Model Component in Current Release and Generated Code from Previous Release

In a previous release, suppose that you developed a model component, generated code for the component, and tested and deployed the generated code. Now, in the current release, you want to add features to the model component and use the model component in system development and code generation. Before you proceed, you can compare the functional behavior of the model component and the generated code from the previous release.

To test the numerical equivalence between the model component and the generated code from the previous release, use Simulink Test. With the Test Manager (Simulink Test), you can perform back-to-back tests and output comparisons:

1. Bring the model component into the current release as a Model block with the **Simulation mode** block parameter set to `Normal`.
2. With the Model block, create a top model that specifies test input data.
3. Import the code generated in the previous release into the current release as a SIL block.
4. With the SIL block, create another top model that specifies the same the test input data.
5. In the Test Manager, create an equivalence test case that runs simulations of the top models and compares outputs.

6 Run the test case and review results.

For more information, see Test Manager examples in "Simulink Test Examples" (Simulink Test).

Note: If you want to compare the behavior of generated code from the current and previous release, in step 1, specify these Model block parameters:

- Set **Simulation mode** to `Software-in-the-loop (SIL)` or `Processor-in-the-loop (PIL)`.
- Set **Code interface** to `Top model`.

See Also

`crossReleaseImport` | `crossReleaseExport` | `sharedCodeUpdate`

Related Examples

- "Cross-Release Shared Utility Code Reuse" on page 33-93

More About

- "Generate Shared Utility Code" (Simulink Coder)
- "Manage the Shared Utility Code Checksum" (Simulink Coder)
- "Generate Shared Utility Code for Fixed-Point Functions" (Simulink Coder)
- "Generate Shared Utility Code for Custom Data Types" (Simulink Coder)

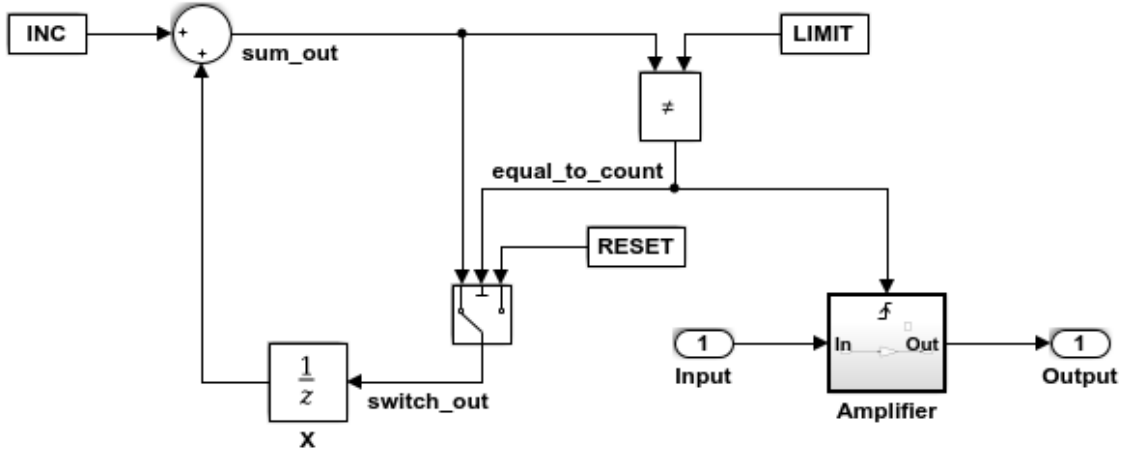
Generate Code Using Simulink® Coder™

This example shows how to select a target for a Simulink® model, generate C code for real-time simulation, and view generated files.

The model represents an 8-bit counter that feeds a triggered subsystem that is parameterized by constant blocks `INC`, `LIMIT`, and `RESET`. `Input` and `Output` represent I/O for the model. The `Amplifier` subsystem amplifies the input signal by gain factor `K`, which updates when signal `equal_to_count` is `true`.

1. Open the model. For example, type the following commands at the MATLAB® command prompt.

```
model='rtwdemo_rtwintro';
open_system(model)
```

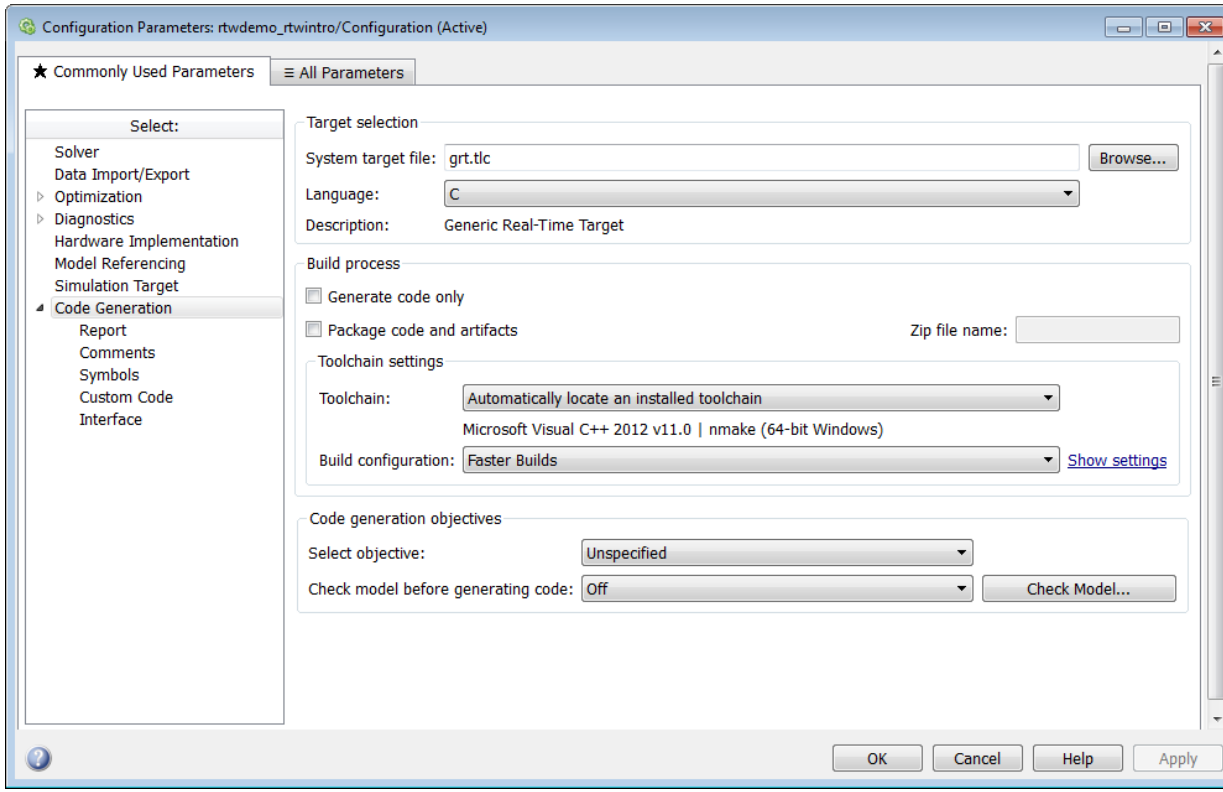


2. Open the Configuration Parameters dialog box from the model editor by clicking **Simulation > Configuration Parameters**.

Alternately, type the following commands at the MATLAB® command prompt.

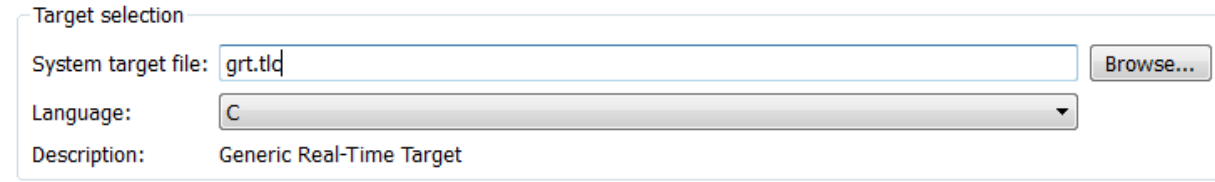
```
cs = getActiveConfigSet(model);
openDialog(cs);
```

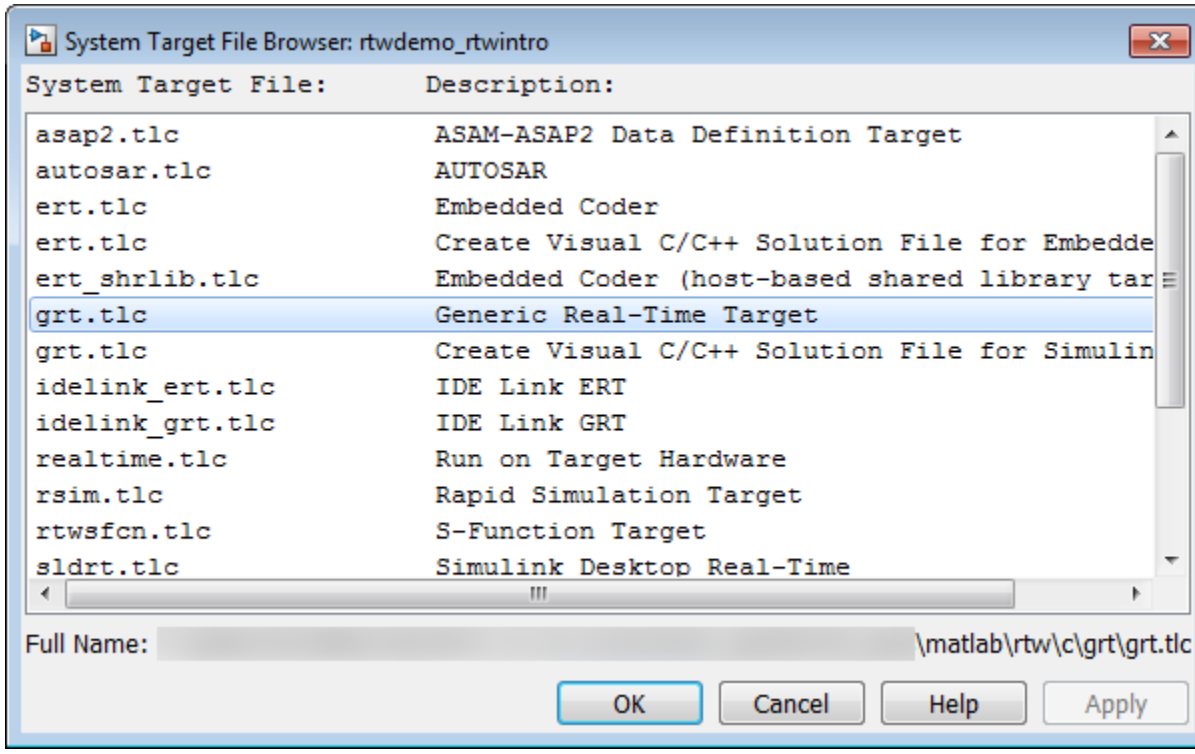
3. Select the **Code Generation** node.



4. In the **Target Selection** pane, click **Browse** to select a target.

You can generate code for a particular target environment or purpose. Some built-in targeting options are provided using system target files, which control the code generation process for a target.





5. Select the **Generic Real-Time (GRT)** target and click **Apply**.

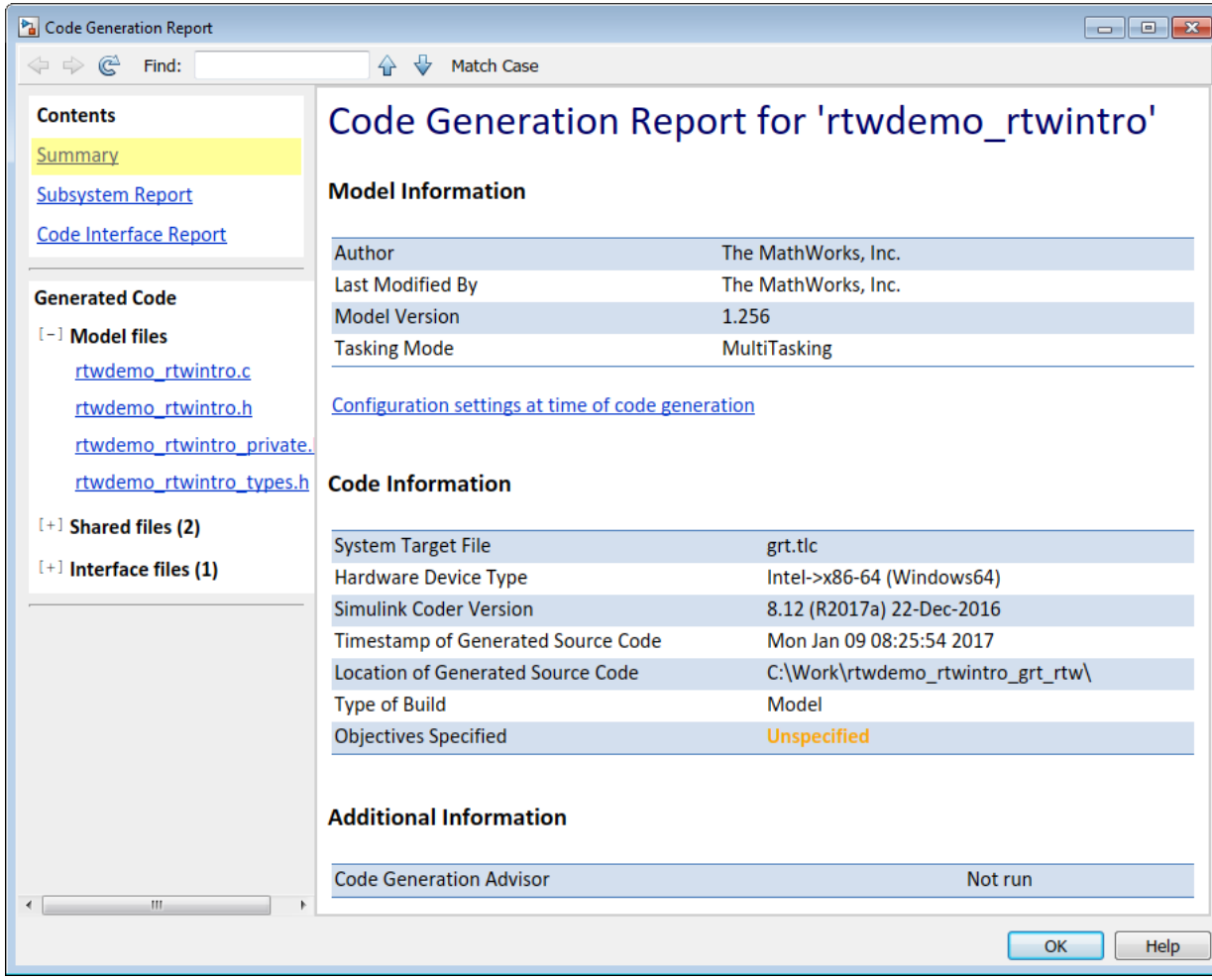
Optionally, in the **Code Generation Advisor** pane set the **Select objective** field to **Execution efficiency** or **Debugging**. Then click **Check model...** to identify and systematically change parameters to meet your objectives.

6. In the model window, initiate code generation and the build process for the model by using any of the following options:

- Click the Build Model button.
- Press **Ctrl+B**.
- Select **Code** > **C/C++ Code** > **Build Model**.
- Invoke the `rtwbuild` command from the MATLAB command line.
- Invoke the `slbuild` command from the MATLAB command line.

7. View the code generation report that appears.

The report includes links to model files such as `rtwdemo_rtwintro.c` and associated utility and header files.



The figure below contains a portion of `rtwdemo_rtwintro.c`

Step function for model: rtwdemo_rtwintro

File: rtwdemo_rtwintro.c

```
1   /* Model step function */
2   void rtwdemo_rtwintro_step(void)
3   {
4     uint8_T rtb_sum_out;
5     boolean_T rtb_equal_to_count;
6
7     /* Sum: '<Root>/Sum' incorporates:
8      *  Constant: '<Root>/INC'
9      *  UnitDelay: '<Root>/X'
10     */
11    rtb_sum_out = (uint8_T)(1U + (uint32_T)rtwdemo_rtwintro_DWork.X);
12
13    /* RelationalOperator: '<Root>/RelOpt' incorporates:
14     *  Constant: '<Root>/LIMIT'
15     */
16    rtb_equal_to_count = (rtb_sum_out != 16);
17
18    /* Outputs for Triggered SubSystem: '<Root>/Amplifier' incorporates:
19     *  TriggerPort: '<S1>/Trigger'
20     */
21    if (rtb_equal_to_count && (rtwdemo_rtwintro_PrevZCSigState.Amplifier_Trig_ZCE
22          != POS_ZCSIG)) {
23      /* Outport: '<Root>/Output' incorporates:
24       *  Gain: '<S1>/Gain'
25       *  Inport: '<Root>/Input'
26       */
27      rtwdemo_rtwintro_Y.Output = rtwdemo_rtwintro_U.Input << 1;
28    }
29
30    rtwdemo_rtwintro_PrevZCSigState.Amplifier_Trig_ZCE = (uint8_T)
31      (rtb_equal_to_count ? (int32_T)POS_ZCSIG : (int32_T)ZERO_ZCSIG);
32
33    /* End of Outputs for SubSystem: '<Root>/Amplifier' */
34
35    /* Switch: '<Root>/Switch' */
36    if (rtb_equal_to_count) {
37      /* Update for UnitDelay: '<Root>/X' */
38      rtwdemo_rtwintro_DWork.X = rtb_sum_out;
39    } else {
40      /* Update for UnitDelay: '<Root>/X' incorporates:
41       *  Constant: '<Root>/RESET'
42       */
43      rtwdemo_rtwintro_DWork.X = 0U;
44    }
45
46    /* End of Switch: '<Root>/Switch' */
47  }
```

8. Close the model.

```
bdclose(model)
rtwdemoclean;
```

Source Code Generation in Embedded Coder

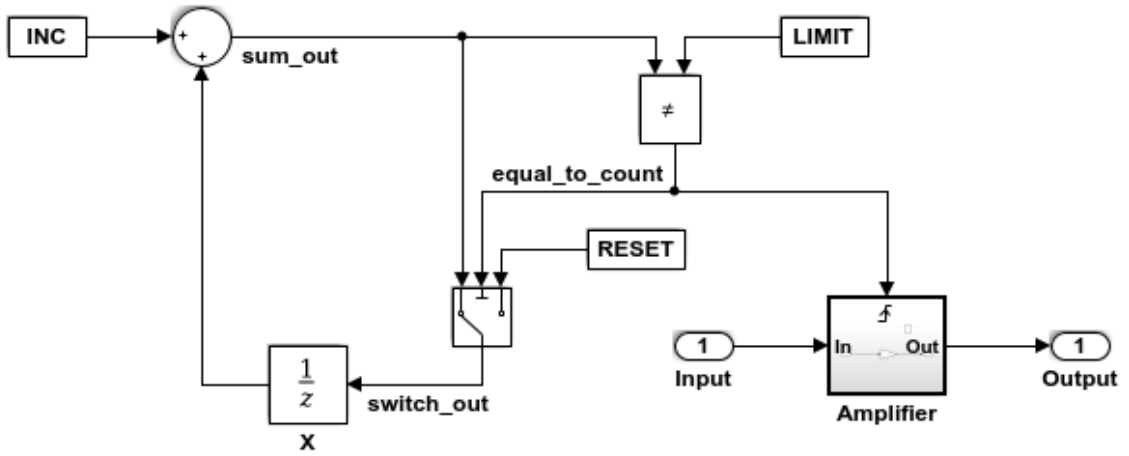
Generate Code Using Embedded Coder®

This example shows how to select a target for a Simulink® model, configure options, generate C code for embedded systems, and view generated files.

The model represents an 8-bit counter that feeds a triggered subsystem that is parameterized by constant blocks `INC`, `LIMIT`, and `RESET`. `Input` and `Output` represent I/O for the model. The `Amplifier` subsystem amplifies the input signal by gain factor `K`, which updates when signal `equal_to_count` is `true`.

1. Open the model.

```
model='rtwdemo_rtwecintro';
open_system(model)
```

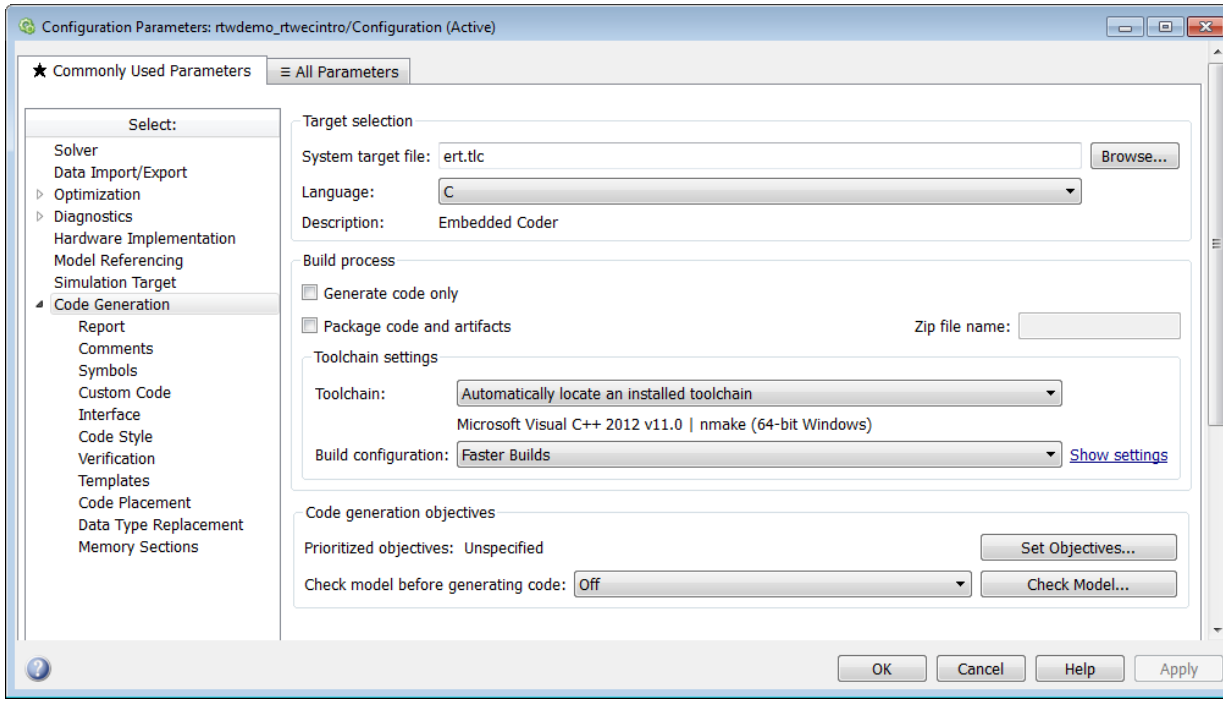


2. Open the **Configuration Parameters** dialog box from the model editor by clicking **Simulation > Model Configuration Parameters**.

Alternately, type the following commands at the MATLAB® command prompt.

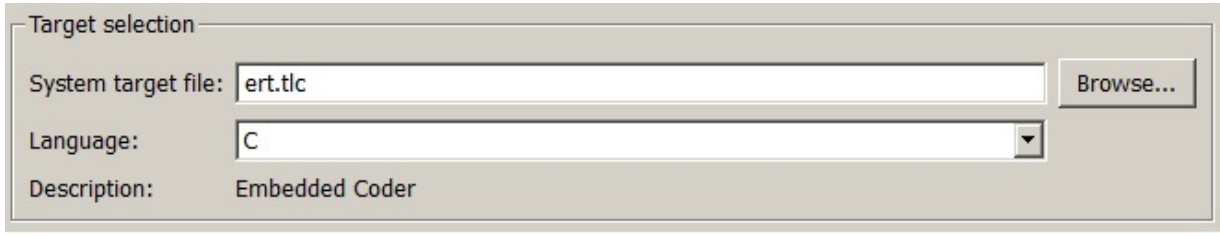
```
cs = getActiveConfigSet(model);
openDialog(cs);
```

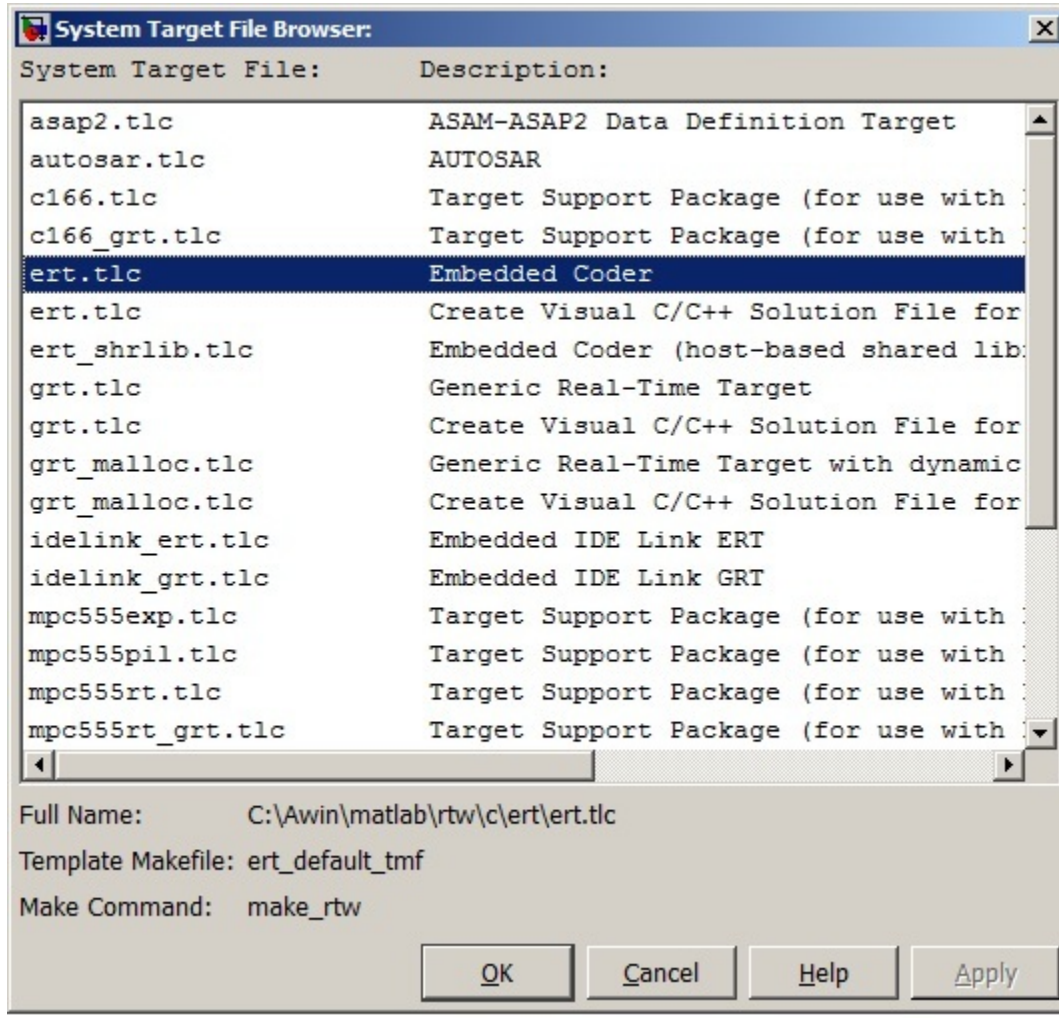
3. Select the **Code Generation** node.



4. In the **Target Selection** pane, click **Browse** to select a target.

You can generate code for a particular target environment or purpose. Some built-in targeting options are provided using system target files, which control the code generation process for a target.


Target selection
System target file:
ert.tlc
Browse...
Language:
C
Description:
Embedded Coder

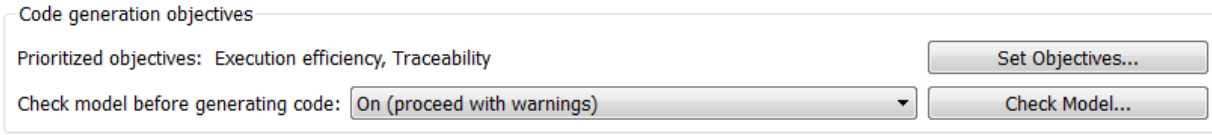


5. Select the **Embedded Real-Time (ERT)** target and click **Apply**.

The ERT target includes a utility to specify and prioritize code generation settings based on your application objectives.

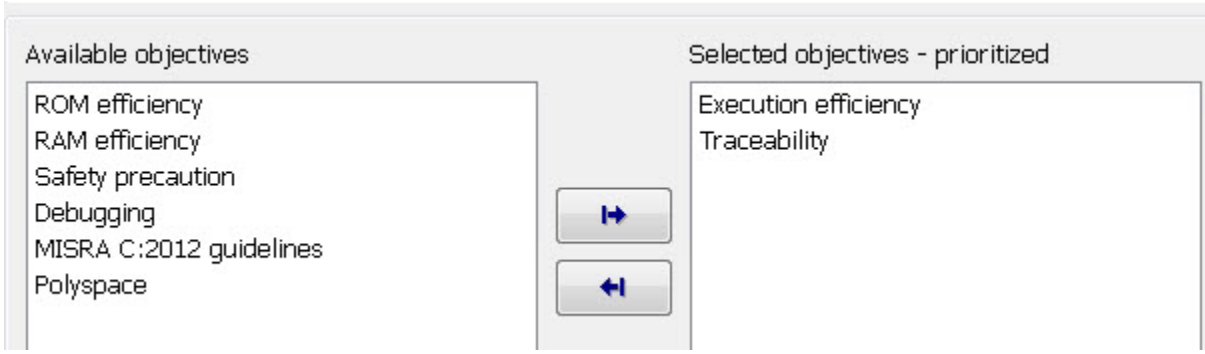
6. In the **Code Generation Advisor** pane, click **Set Objectives**.

You can set and prioritize objectives for the generated code. For example, while code traceability might be a very important criterion for your application, you might not want to prioritize it at the cost of code execution efficiency.



7. In the **Set Objectives** pane, select **Execution efficiency** and **Traceability**. Click **OK**.

You can select and prioritize a combination of objectives before generating code.

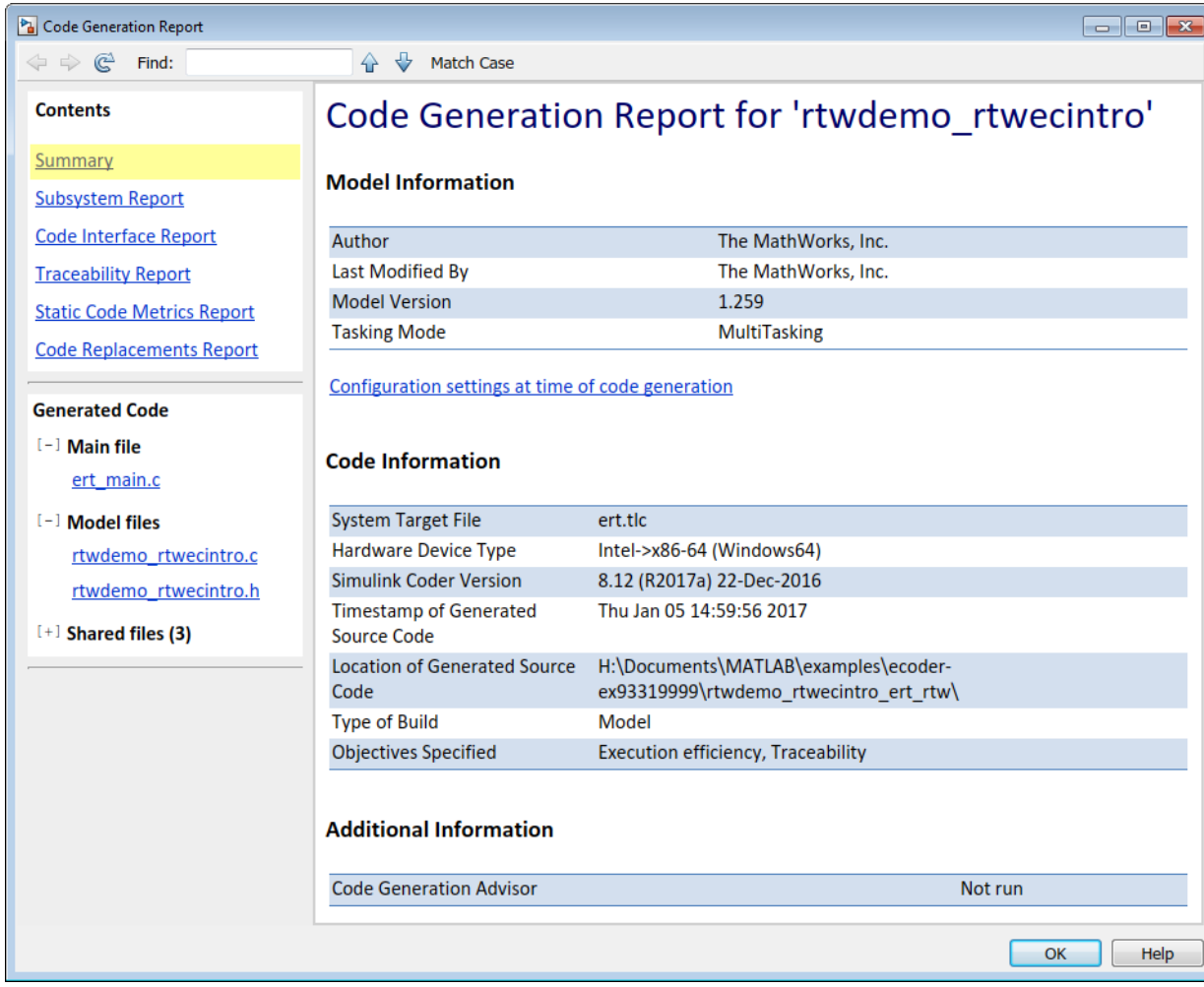


8. In the model window, initiate code generation and the build process for the model by using any of the following options:

- Click the Build Model button.
- Press **Ctrl+B**.
- Select **Code** > **C/C++ Code** > **Build Model**.
- Invoke the `rtwbuild` command from the MATLAB command line.
- Invoke the `slbuild` command from the MATLAB command line.

9. View the code generation report that appears.

The report includes `rtwdemo_rtwecintro.c`, associated utility and header files, and traceability and validation reports.



The figure below contains a portion of `rtwdemo_rtwecintro.c`

Step function for model: rtwdemo_rtwecintro

File: rtwdemo_rtwecintro.c

```
1   /* Model step function */
2   void rtwdemo_rtwecintro_step(void)
3   {
4     boolean_T rtb_equal_to_count;
5
6     /* Sum: '<Root>/Sum' incorporates:
7      *  Constant: '<Root>/INC'
8      *  UnitDelay: '<Root>/X'
9      */
10    rtDWork.X = (uint8_T)(1U + (uint32_T)rtDWork.X);
11
12    /* RelationalOperator: '<Root>/RelOpt' incorporates:
13     *  Constant: '<Root>/LIMIT'
14     */
15    rtb_equal_to_count = (rtDWork.X != 16);
16
17    /* Outputs for Triggered SubSystem: '<Root>/Amplifier' incorporates:
18     *  TriggerPort: '<S1>/Trigger'
19     */
20    if (rtb_equal_to_count && (rtPrevZCSigState.Amplifier_Trig_ZCE != POS_ZCSIG))
21    {
22      /* Outport: '<Root>/Output' incorporates:
23       *  Gain: '<S1>/Gain'
24       *  Inport: '<Root>/Input'
25       */
26      rtY.Output = rtU.Input << 1;
27    }
28
29    rtPrevZCSigState.Amplifier_Trig_ZCE = (uint8_T)(rtb_equal_to_count ? (int32_T)
30      POS_ZCSIG : (int32_T)ZERO_ZCSIG);
31
32    /* End of Outputs for SubSystem: '<Root>/Amplifier' */
33
34    /* Switch: '<Root>/Switch' */
35    if (!rtb_equal_to_count) {
36      /* Update for UnitDelay: '<Root>/X' incorporates:
37       *  Constant: '<Root>/RESET'
38       */
39      rtDWork.X = 0U;
40    }
41
42    /* End of Switch: '<Root>/Switch' */
43  }
```

10. Close the model.

```
bdclose(model)
rtwdemoclean;
```

More About

- "Generate Code with the Quick Start Tool" on page 34-10

Generate Code with the Quick Start Tool

The Quick Start tool helps you prepare a model for generating readable, efficient code. To start the tool, from the model window, select **Code > C/C++ > Embedded Coder Quick Start**.

After you start the tool, you must answer these questions about the code that you want to generate:

- What is the model or subsystem for code generation?
- What is the type of code output for your generated code?
- What is the target hardware processor type?
- What is your primary code generation objective?

The tool validates your choices against the model and presents the parameter changes required to generate code. If you choose to generate code, the tool applies the changes to your configuration set and generates the code. After code generation, you can view the code generation report and find information on building, customizing, optimizing, and packaging the code.

Quick Start Model Analysis

At each step of the Quick Start process, the tool validates your model against your selections. The tool determines if there are model conditions that prevent you from proceeding with code generation. During the analysis step, the tool must also examine your model or subsystem for answers to the following questions. The answers help determine the best configuration for the deployment of your code.

How many sample rates are in your system?

The Quick Start tool evaluates your model to determine the number of periodic sample rates in your system.

Single rate	Your model has only one periodic sample rate. The generated code has a single-entry point function that runs at the time interval of the sample rate.
Multirate	Your model has more than one periodic sample rate. It is possible that the generated code does not execute at the same time intervals. Following the analysis step, you can choose to generate a single-entry point function for each

	of the sample rates, or generate a different entry point function for each sample rate. If you choose to generate multitasking code, the code generator produces multiple entry-point functions. These functions run as multiple tasks. Each entry point function is called at an interval defined by the sample rate that is configured in the model.

Note: If your model contains an asynchronous rate, an additional entry point function is generated to run at the specific interrupt time.

For more information about sample rates, see "Time-Based Scheduling and Code Generation" (Simulink Coder).

Does your system contain continuous states?

The Quick Start tool evaluates your model for continuous blocks to determine the correct solver to use.

No	If your system does not contain continuous states, the Quick Start tool configures your model to use a fixed-step discrete solver for code generation if you have not selected one.
Yes	If your system does contain continuous states, the Quick Start tool configures your model to use a fixed-step continuous solver for code generation if you have not selected one. It also selects the `SupportContinuous` configuration parameter.

For more information on solvers, see "About Solvers" (Simulink).

Did you configure your system for export function calls?

The Quick Start tool evaluates your model to see if scheduler code must be generated.

No	If you did not configure your system for export function calls, the generated code includes code for the system algorithm and scheduler code.
Yes	If you configured your system for export function calls, the generated code includes code for the system algorithm. You can manually write the scheduler code or generate it from other models.

For more information, see “Export-Function Models” (Simulink).

Does your system contain referenced models?

The Quick Start tool evaluates your model to see if it depends on code from other models.

No	If your system does not contain referenced models, the generated code does not depend on code from other models.
Yes	If your system contains referenced models, the generated code for your model depends on other modules generated from referenced models. The code generator can optimize the generated code because it is aware of the relationship between your model and the referenced models.

For more information, see “Code Generation of Referenced Models” (Simulink Coder).

Configuration Parameter Changes for Models with a Configuration Reference

To apply configuration parameter changes to a model with an active configuration reference, the Quick Start tool:

- Creates a new `Simulink.ConfigSet` object, `QuickStart_`*`timestamp`*, in the workspace or data dictionary that contains the original configuration set. The new object is a copy of the original configuration set with the parameter changes applied.
- Creates a new `Simulink.ConfigSetRef` object that points to the new configuration set object.
- Attaches the new configuration reference to the model and makes it the active configuration.

To restore the original configuration set, activate the original `Simulink.ConfigSetRef` object.

Note: If the Quick Start tool creates the new configuration set object in the MATLAB workspace, you must save it to preserve the configuration set after the MATLAB session ends. For more information, see “Save a Configuration Set” (Simulink).

Next Steps

After you have generated code by using Quick Start, possible next steps are:

- “Open Code Generation Report” on page 35-8
- “Code Appearance”
- “Build Process”
- “Application Objectives Using Code Generation Advisor” (Simulink Coder)
- “Manage a Configuration Set” (Simulink)
- “Configure Model and Generate Code” (Simulink Coder)
- “Relocate Code to Another Development Environment” (Simulink Coder)

Manage File Packaging of Generated Code Modules

The code generator produces code modules. The file packaging configuration controls where the code generator places code into code modules and header files.

To locate and examine the generated code files, use the HTML code generation report. The code generation report provides a table of hyperlinks that you click to view the generated code in the MATLAB Help browser. For more information, see "Traceability in Code Generation Report" on page 35-15.

In this section...

Generated Code Modules

The code generator creates a build folder in your working folder to store generated source code. The build folder contains object files, a makefile, and other files created during the code generation process. The default name of the build folder is *`model`*`_ert_rtw`.

Code Modules and Header Files Affected by File Packaging summarizes the structure of source code that the code generator produces.

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

- File packaging formats: Manage the number of source files generated for your model. In the Configuration Parameter dialog box, on the **Code Generation > Code Placement** pane, specify the **File packaging format** parameter. For more information, see "Customize Generated Code Modules" on page 34-17.
- Nonvirtual subsystem code generation: Instruct the code generation software to generate separate functions within separate code files for nonvirtual subsystems. You can control the names of the functions and of the code files. For further information, see "Code Generation of Subsystems" (Simulink Coder).
- Custom storage classes: Use custom storage classes to partition generated data structures into different files based on file names that you specify. For further information, see "Introduction to Custom Storage Classes" on page 23-2.

- Module Packaging Features (MPF): Direct the generated code into a required set of `.c` or `.cpp` and `.h` files, and control the internal organization of the generated files. For details, see "Data, Function, and File Definition".

Code Modules and Header Files Affected by File Packaging

File	Description
`model``.c` or `.cpp`	Contains entry points for code implementing the model algorithm (for example, *`model`*`_step`, *`model`*`_initialize`, and *`model`*`_terminate`).
`model``_private.h`	Contains local macros and local data that the model and subsystems require. This file is included in the `model`.c file as a `#include` statement. You do not need to include *`model`*`_private.h` when interfacing handwritten code to the generated code of a model.
`model``.h`	Declares model data structures and a public interface to the model entry-points and data structures. Provides an interface to the real-time model data structure (*`model`*`_M`) with accessor macros. The code generator: • Produces a separate header file for each Simulink Function block in a model. • Includes *`model`*`.h` in the subsystem `.c` or `.cpp` files of a model. If you interface handwritten code to generated code for one or more models, include *`model`*`.h` for each of those models.
`model``_data.c` or `.cpp`	Contains (if conditionally generated) the declarations for the parameters data structure, the constant block I/O data structure, and any zero representations for the model structure data types. If the model does not use these data structures and zero representations, *`model`*`_data.c` or `.cpp` is not generated. 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. Function declarations of reusable functions can require these declarations. Provides type definitions for user-defined types that the model uses.

File	Description
`rtwtypes.h`	Defines data types, structures, and macros required by generated code. For more information, see "Control Placement of rtwtypes.h for Shared Utility Code" (Simulink Coder).
`multiword_types.h`	Contains type definitions for wide data types and their chunks. File is generated when multiword data types are used or when you select one or more of these configuration parameters: • **All Parameters > MAT-file logging** • **Code Generation > Interface > External mode**
`model_reference_types.h`	Contains type definitions for timing bridges. File is generated for a model reference target or a model containing model reference blocks.
`builtin_typeid_types.h`	Defines an enumerated type corresponding to built-in data types. File is generated when your model contains a Stateflow chart that uses messages or when you select one or more of these configuration parameters: • **All Parameters > MAT-file logging** • Any C API option at **Code Generation > Interface**
`zero_crossing_types.h`	Contains zero-crossing definitions for models with triggered subsystems where the trigger is `rising`, `falling`, or `either`. File is generated only if required by the model.
`ert_main.c` or `.cpp`	(optional file) If the **Generate an example main program** option is on (default), this file is generated. For more information, see "Generate an example main program".
`rtmodel.h`	(optional file) If the **Generate an example main program** option is off, this file is generated. For more information, see "Generate an example main program". Contains `#include` directives required by the `rt_main.c` or `rt_cppclass_main.cpp` static main program module. Includes `rtmodel.h` to access model-specific data structures and entry points, because the static main program module is not created at code generation time. For more information, see "Static Main Program Module" on page 49-10.

File	Description
*model*_capi.c or .cpp *model*_capi.h	(optional file) Provides data structures that enable a running program to access model signals, states, and parameters without external mode. To learn how to generate and use the *model*_capi.c or .cpp and .h files, see "Exchange Data Between Generated and External Code Using C API" (Simulink Coder) in the Simulink Coder documentation.

User-Written Code Modules

Code that you write to interface with generated model code usually includes a customized main module. Base this module on a main program provided by the code generation software. This customized main module can also include interrupt handlers, device driver blocks and other S-functions, and other supervisory or supporting code.

Establish a working folder for your own code modules. Put your working folder on the MATLAB path. At minimum, inform the build process about the location of your source and object files with **Additional build information** in the **Code Generation > Custom Code** pane. Your development process could require generating code for a particular microprocessor or development board and deploying the code on target hardware with a cross-development system. To accomplish these goals, make more extensive modifications to the ERT-based system target file.

For information on how to customize your ERT-based system target file for your production requirements, see "Target Development" (Simulink Coder).

Customize Generated Code Modules

A configuration parameter is available to specify how the code generator packages generated source code into files. The configuration parameter **File packaging format** options are located in the Configuration Parameter dialog box, on the **Code Generation > Code Placement** pane, in the **Code packaging** section. The options are `Modular`, `Compact (with separate data file)`, and `Compact`. The table describes the files generated for each file packaging format and the files that have been removed.

Generated Files According to File Packaging Format

File Packaging Format	Generated Files	Removed Files
`Modular` (default)	*model*.c	None

File Packaging Format	Generated Files	Removed Files
	subsystem files (optional) *model*.h *model*_types.h *model*_private.h *model*_data.c (conditional)	
Compact (with separate data file)	*model*.c *model*.h *model*_data.c (conditional)	*model*_private.h *model*_types.h (conditional, see Removed Files According to File Packaging Format)
Compact	*model*.c *model*.h	*model*_data.c *model*_private.h *model*_types.h (conditional, see Removed Files According to File Packaging Format)

The table describes content placement from the removed files.

Removed Files According to File Packaging Format

Removed File	Generated Content In File
*model*_private.h	*model*.c and *model*.h
*model*_types.h	*model*.h
*model*_data.c	*model*.c

You can specify a different file packaging format for each referenced model.

The **Configuration Parameter > Code Generation > Interface > Shared code placement** selection interacts with file packaging operations. If you specify **Shared code placement** as `Shared location`, the code generator generates separate files for utility code in a shared location, regardless of the file packaging format. If you specify

the **Shared code placement** as `Auto`, the code generator generates code for utilities according to the file packaging format selection.

- `Modular`: Some shared utility files are in the build folder.
- `Compact (with separate data file)`: Utility code is generated in *`model`*`.c`.
- `Compact`: Utility code is generated in *`model`*`.c`.

File packaging formats `Compact` and `Compact (with separate data file)` generate *`model`*`_types.h` for models containing:

- A Model Variants block or a Variant Subsystem block. The *`model`*`_types.h` file includes preprocessor directives defining the variant objects associated with a variant block.
- Custom storage classes generating a separate header file.

File packaging formats `Compact` and `Compact (with separate data file)` are not compatible with:

- A model containing a subsystem, which is configured to generate separate source files
- A model containing a noninlined S-function
- A model for which **Shared code placement** is set to `Auto`, which uses data objects for which **Data scope** is set to `Exported`

More About

- "Manage Build Process Folders" on page 33-37
- "Manage Build Process Files" on page 33-42
- "Manage Build Process File Dependencies" on page 33-52

Generate Reentrant Code from Top-Level Models

To generate reentrant multi-instance code from a model, select `Reusable function` code interface packaging. When you select the `Reusable function` code interface for an ERT-based model:

- By default, the generated *`model`*`.c` source file does not contain an allocation function that dynamically allocates model data for each instance of the model. Use the **Use dynamic memory allocation for model initialization** option to control whether an allocation function is generated.
- The generated code passes the real-time model data structure in, by reference, as an argument to *`model`*`_step` and the other model entry point functions.
- The real-time model data structure is exported with the *`model`*`.h` header file.
- By default, root-level input and output arguments are passed to the reusable model entry-point functions as individual arguments. Use the **Pass root-level I/O as** parameter to control whether root-level input and output arguments are passed. This selection chooses whether this I/O is included in the real-time model data structure that is passed to the functions, passed as individual arguments, or passed as references to an input structure and an output structure.

To configure an ERT-based model to generate reusable, reentrant code:

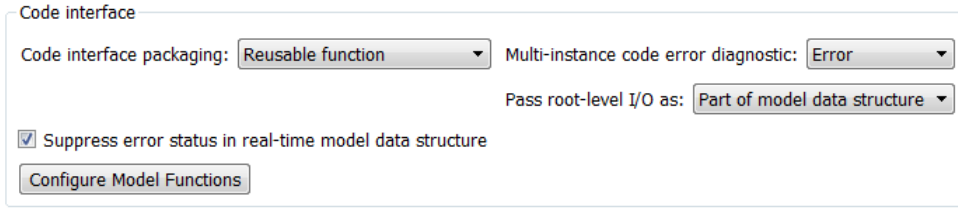
1. In the **Code Generation > Interface** pane of the Configuration Parameters dialog box, set **Code interface packaging** (Simulink Coder) to the value `Reusable function`. This action enables the parameters **Multi-instance code error diagnostic**, **Pass root-level I/O as**, and **All Parameters > Use dynamic memory allocation for model initialization**.
2. Examine the setting of **Multi-instance code error diagnostic** (Simulink Coder). Leave the parameter at its default value `Error` unless you have a specific need to alter the severity level for diagnostics displayed when a model violates requirements for generating multi-instance code.
3. Configure **Pass root-level I/O as** (Simulink Coder) to control how root-level model input and output are passed to `model_step` and the other generated model entry-point functions.

   When you set **Code interface packaging** to `Reusable function`, model data (such as block I/O, DWork, and parameters) is packaged into the real-time model data structure, and the model structure is passed to the model entry-point functions. If you set **Pass root-level I/O as** to `Part of model data structure`, the

root-level model input and output also are packaged into the real-time model data structure.

4. If you want the generated model code to contain a function that dynamically allocates memory for model instance data, on the **All Parameters** tab, select the option **Use dynamic memory allocation for model initialization** (Simulink Coder). If you do not select this option, the generated code statically allocates memory for model data structures.
5. Generate model code.
6. Examine the model entry-point function interfaces in the generated files and the HTML code generation report. For more information about generating and calling model entry-point functions, see "Entry-Point Functions and Scheduling" (Simulink Coder).

For an example of a model configured to generate reusable, reentrant code, open the example model rtwdemo_reusable. Click the button **View Interface Configuration** and examine the **Code interface** parameters on the **Code Generation > Interface** pane.



More About

- "Entry-Point Functions and Scheduling" (Simulink Coder)

Report Generation in Embedded Coder

Reports for Code Generation

In this section...

The code generator software produces an HTML code generation report so that you can view and analyze the generated code. When your model is built, the code generation process produces an HTML file that is displayed in an HTML browser or in the Model Explorer. The code generation report includes:

- The **Summary** section that contains model and code information, including **Author**, **Tasking Mode**, **System Target File**, **Hardware Device Type**, and code generation objectives information. The **Configuration settings at the time of code generation** link opens a noneditable view of the Configuration Parameters dialog box. The dialog box shows the Simulink model settings at the time of code generation, including TLC options.
- The **Subsystem Report** section that contains information on nonvirtual subsystems in the model.
- In the **Generated Files** section on the **Contents** pane, you can click the names of source code files generated from your model to view their contents in a MATLAB Web browser window. In the displayed source code, global variables are hypertext that links to their definitions.

For an example, see "Generate a Code Generation Report" on page 35-5.

If you have a Simulink Report Generator license, you can document your code generation project in multiple formats, including HTML, PDF, RTF, Microsoft Word, and XML. For an example of how to create a Microsoft Word report, see "Document Generated Code with Simulink Report Generator" on page 35-52.

HTML Code Generation Report Location

The default location for the code generation report files is in the `html` subfolder of the build folder, *`model_target`*`_rtw/html/`. *`target`* is the name of the **System target file** specified on the **Code Generation** pane. The default name for the top-level HTML report file is *`model`*`_codegen_rpt.html` or *`subsystem`*`_codegen_rpt.html`. For

more information on the location of the build folder, see “Manage Build Process Folders” (Simulink Coder).

HTML Code Generation Report for Referenced Models

To generate a code generation report for a top model and code generation reports for each referenced model, you need to specify the **Create code generation report** on the **Code Generation > Report** pane for the top model and each referenced model. You can open the code generation report of a referenced model in one of two ways:

- From the top-model code generation report, you can access the referenced model code generation report by clicking a link under **Referenced Models** in the left navigation pane. Clicking a link opens the code generation report for the referenced model in the browser. To navigate back to the top model code generation report, use the **Back** button at the top of the left navigation pane.
- From the referenced model diagram window, select **Code > C/C++ Code > Code Generation Report > Open Model Report**.

For more information, see “Generate Code for Referenced Models” (Simulink Coder)

HTML Code Generation Report Extensions

If you have an Embedded Coder license, the code generator enhances the HTML code generation report. Configure your model to include the following sections in the report:

- The **Code Interface Report** section provides information about the generated code interface, including model entry-point functions and input/output data. For more information, see “Analyze the Generated Code Interface” on page 35-21.
- The **Traceability Report** section allows you to account for **Eliminated / Virtual Blocks** that are untraceable versus the listed **Traceable Simulink Blocks / Stateflow Objects / MATLAB Scripts**. This provides a complete mapping between model elements and code. For more information, see “Customize Traceability Reports” on page 61-29.
- The **Static Code Metrics Report** section provides statistics of the generated code. Metrics are estimated from static analysis of the generated code. For more information, see “Static Code Metrics” on page 35-34.
- The **Code Replacements Report** section allows you to account for code replacement library (CRL) functions that were used during code generation, providing a mapping between each replacement instance and the Simulink block that triggered the

replacement. For more information, see “Analyze Code Replacements in Generated Code” on page 35-50.

- The model Web view displays an interactive model diagram within the code generation report and supports traceability between the source code and the model. Therefore, you can share your model and generated code outside of the MATLAB environment. For more information, see “Web View of Model in Code Generation Report” on page 35-17.

On the **Contents** pane, in the **Generated Files** section, you can click the names of source code files generated from your model to view their contents in a MATLAB Web browser window. In the displayed source code:

- If you enable code-to-model traceability, hyperlinks within the displayed source code navigate to the blocks or subsystems from which the code is generated. For more information, see “Traceability in Code Generation Report” on page 35-15 and “Trace Code to Model Objects by Using Hyperlinks” on page 61-6.
- If you enable model-to-code traceability, you can navigate to the generated code for a block in the model. For more information, see “Trace Model Objects to Generated Code” on page 61-8.
- If you set the **Code coverage tool** parameter on the **Code Generation > Verification** pane, you can view the code coverage data and annotations. For more information, see “Configure Code Coverage with Third-Party Tools” on page 67-10.
- If you select the **Static code metrics** check box on the **Code Generation > Report** pane, you can view code metrics information and navigate to code definitions and declarations in the generated code. For more information, see “View Static Code Metrics and Definitions Within the Generated Code” on page 35-36.

Related Examples

- “Traceability in Code Generation Report” on page 35-15

Generate a Code Generation Report

To generate a code generation report when the model is built:

1. In the Simulink Editor, select **Code > C/C++ Code > Code Generation Report > Options**. The Configuration Parameters dialog box opens with the **Code Generation > Report** pane visible.
2. Select the **Create code generation report** (Simulink Coder) parameter.
3. If you want the code generation report to automatically open after generating code, select the **Open report automatically** (Simulink Coder) parameter (which is enabled by selecting **Create code generation report**).
4. Generate code.

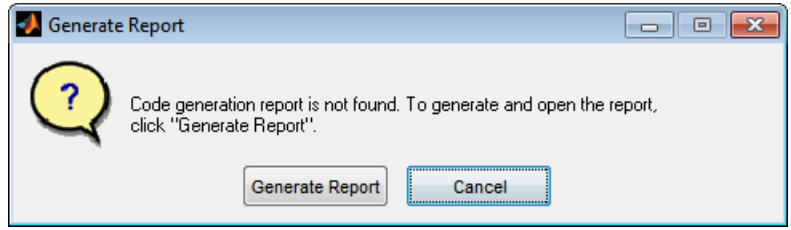
The build process writes the code generation report files to the `html` subfolder of the build folder (see "HTML Code Generation Report Location" on page 35-2). Next, the build process automatically opens a MATLAB Web browser window and displays the code generation report.

To open an HTML code generation report at any time after a build, see "Open Code Generation Report" on page 35-8 and "Generate Code Generation Report After Build Process" on page 35-6.

Generate Code Generation Report After Build Process

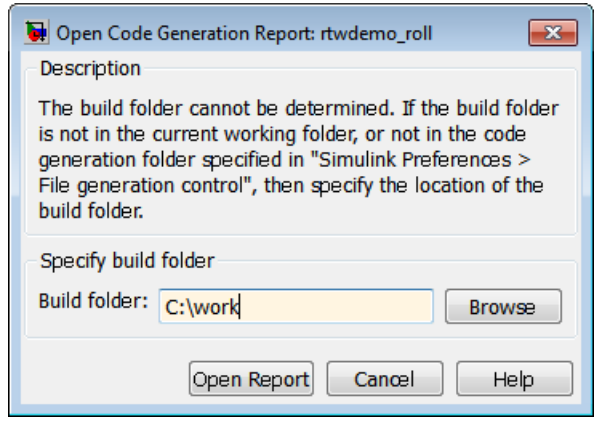
After generating code, if you did not configure your model to create a code generation report, you can generate a code generation report without rebuilding your model.

1. In the model diagram window, select **Code > C/C++ Code > Code Generation Report > Open Model Report**.
2. If your current working folder contains the code generation files the following dialog opens.



Click **Generate Report**.

3. If the code generation files are not in your current working directory, the following dialog opens.



Enter the full path of the build folder for your model, `../model_target_rtw` and click **Open Report**.

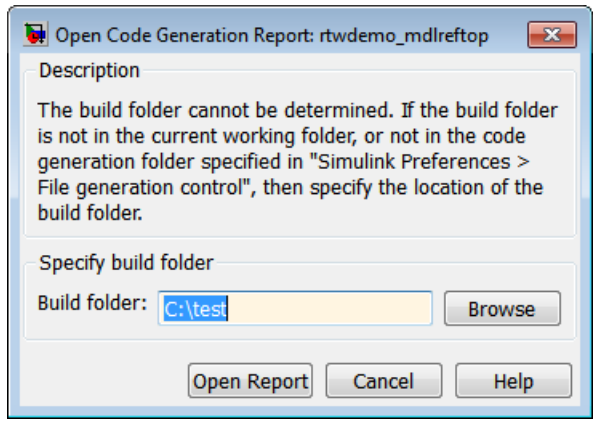
The software generates a report, *model*_codgen_rpt.html, from the code generation files in the build folder you specified.

Note: An alternative method for generating the report after the build process is complete is to configure your model to generate a report and build your model. In this case, the software generates the report without regenerating the code.

Open Code Generation Report

You can refer to existing code generation reports at any time. If you generated a code generation report, in the Simulink Editor, you can open the report by selecting the menu option **Code > C/C++ Code > Code Generation Report > Open Model Report**. If you are opening a report for a subsystem, select **Open Subsystem Report**. A Simulink Coder license is required to view the code generation report. An Embedded Coder license is required to view a code generation report enhanced with Embedded Coder features.

If your current working folder does not contain the code generation files and the code generation report, the following dialog box opens:



Enter the full path of the build folder for your model, `../model_target_rtw` and click **Open Report**.

Alternatively, you can open the code generation report (`model_codegen_rpt.html` or `subsystem_codegen_rpt.html`) manually into a MATLAB Web browser window, or in another Web browser. For the location of the generated report files, see "HTML Code Generation Report Location" on page 35-2.

Limitation

After building your model or generating the code generation report, if you modify legacy or custom code, you must rebuild your model or regenerate the report for the code generation report to include the updated legacy source files. For example, if you modify your legacy code, and then use the **Code > C/C++ Code > Code Generation**

Report > Open Model Report menu to open an existing report, the software does not check if the legacy source file is out of date compared to the generated code. Therefore, the code generation report is not regenerated and the report includes the out-of-date legacy code. This issue also occurs if you open a code generation report using the `coder.report.open` function.

To regenerate the code generation report, do one of the following:

- Rebuild your model.
- Generate the report using the `coder.report.generate` function.

Generate Code Generation Report Programmatically

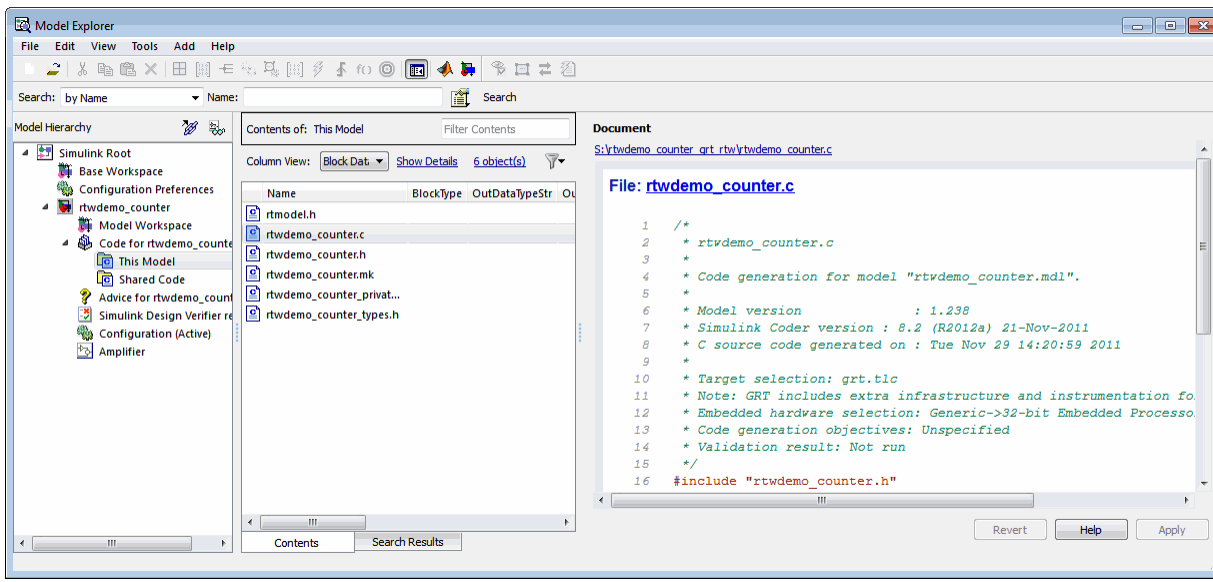
At the MATLAB command line, you can generate, open, and close an HTML Code Generation Report with the following functions:

- `coder.report.generate` generates the code generation report for the specified model.
- `coder.report.open` opens an existing code generation report.
- `coder.report.close` closes the code generation report.

View Code Generation Report in Model Explorer

After generating an HTML code generation report, you can view the report in the right pane of the Model Explorer. You can also browse the generated files directly in the Model Explorer.

When you generate code, or open a model that has generated code for its current target configuration in your working folder, the **Hierarchy** (left) pane of Model Explorer contains a node named **Code for *`model`***. Under that node are other nodes, typically called `This Model` and `Shared Code`. Clicking `This Model` displays in the **Contents** (middle) pane a list of generated source code files in the build folder of that model. The next figure shows code for the `rtwdemo_counter` model.



In this example, the file `S:/rtwdemo_counter_grt_rtw/rtwdemo_counter.c` is being displayed. To view a file in the **Contents** pane, click it once.

The views in the **Document** (right) pane are read only. The code listings there contain hyperlinks to functions and macros in the generated code. Clicking the file hyperlink opens that source file in a text editing window where you can modify its contents.

If an open model contains Model blocks, and if generated code for these models exists in the current `slprj` folder, nodes for the referenced models appear in the **Hierarchy** pane

one level below the node for the top model. Such referenced models do not need to be open for you to browse and read their generated source files.

If the code generator produces shared utility code for a model, a node named `Shared Code` appears directly under the **This Model** node. It collects source files that exist in the `./slprj/`*`target`*`/_sharedutils` subfolder.

Note You cannot use the **Search** tool built into Model Explorer toolbar to search generated code displayed in the Code Viewer. On PCs, typing **Ctrl+F** when focused on the **Document** pane opens a Find dialog box that you can use to search for text in the currently displayed file. You can also search for text in the HTML report window, and you can open the files in the editor.

Package and Share the Code Generation Report

In this section...

Package the Code Generation Report

To share the code generation report, you can package the code generation report files and supporting files into a zip file for transfer. The default location for the code generation report files is in two folders:

- `/slprj`
- `html` subfolder of the build folder, *`model_target`*`_rtw`, for example `rtwdemo_counter_grt_rtw/html`

To create a zip file from the MATLAB command window:

1. In the Current Folder browser, select the two folders:
   - `/slprj`
   - Build folder: *`model_target`*`_rtw`
2. Right-click to open the context menu.
3. In the context menu, select **Create Zip File**. A file appears in the Current Folder browser.
4. Name the zip file.

Alternatively, you can use the MATLAB `zip` command to zip the code generation report files:

```
zip('myzip',{'slprj','rtwdemo_counter_grt_rtw'})
```

Note: If you need to relocate the static and generated code files for a model to another development environment, such as a system or an integrated development environment (IDE) that does not include MATLAB and Simulink products, use the code generator pack-and-go utility. For more information, see "Relocate Code to Another Development Environment" (Simulink Coder).

View the Code Generation Report

To view the code generation report after transfer, unzip the file and save the two folders at the same folder level in the hierarchy. Navigate to the *model_target*_rtw/html/ folder and open the top-level HTML report file named *model*_codgen_rpt.html or *subsystem*_codegen_rpt.html in a Web browser.

Traceability in Code Generation Report

This example shows how to create an HTML code generation report which includes links to trace between the source code and the Simulink model window.

1. With your ERT-based model open, open the Configuration Parameters dialog box or Model Explorer and navigate to the **Code Generation > Report** pane.
2. Select **Create code generation report** if it is not already selected. By default, **Open report automatically** and **Code-to-model** on the **All Parameters** tab are selected. **Model-to-code** is not selected.
3. Select the **Model-to-code** parameter on the **All Parameters** tab.
4. If your model contains referenced models and you want to enable traceability for the referenced model's code generation report, repeat steps 2–3 for each referenced model.
5. Press **Ctrl+B** to generate code for your model. The build process opens the code generation report in a MATLAB Web browser.
6. In the left navigation pane, select a source code file. The source code and line numbers in the right pane contain hyperlinks to blocks in the model.
7. Click a code or line number hyperlink. The model diagram window displays and highlights the corresponding block or blocks in the model.
8. To highlight the generated code for a block in your Simulink model, right-click the block and select **C/C++ Code > Navigate to C/C++ Code**. This selection highlights the generated code for the block in the HTML code generation report.
9. If you have a referenced model in your model, in the left navigation pane, below **Reference Models**, click the link to a referenced model. The code generation report for the referenced model is now displayed in the window.
10. In the left navigation pane, click the **Back** button to go back to the previous code generation report.

Related Examples

- "Trace Model Objects to Generated Code" on page 61-8
- "Trace Code to Model Objects by Using Hyperlinks" on page 61-6
- "Trace Stateflow Objects in Generated Code" on page 61-10

More About

- "What Is Code Tracing?" on page 61-2

- “Traceability Limitations” on page 61-32

Web View of Model in Code Generation Report

In this section...

About Model Web View

To review and analyze the generated code, it is helpful to navigate between the code and model. You can include a Web view of the model within the HTML code generation report. You can then share your model and generated code outside of the MATLAB environment. When you generate the report, the Web view includes the block diagram attributes displayed in the Simulink Editor, such as, block sorted execution order, signal properties, and port data types.

A Simulink Report Generator license is required to include a Web view (Simulink Report Generator) of the model in the code generation report.

Browser Requirements for Web View

Web view requires a Web browser that supports Scalable Vector Graphics (SVG). Web view uses SVG to render and navigate models.

You can use the following Web browsers:

- Mozilla Firefox Version 1.5 or later, which has native support for SVG. To download the Firefox browser, go to `www.mozilla.com/`.
- The Microsoft Internet Explorer® Web browser with the Adobe® SVG Viewer plug-in. To download the Adobe SVG Viewer plug-in, go to `www.adobe.com/svg/`.
- Apple Safari Web browser

Generate HTML Code Generation Report with Model Web View

This example shows how to create an HTML code generation report which includes a Web view of the model diagram.

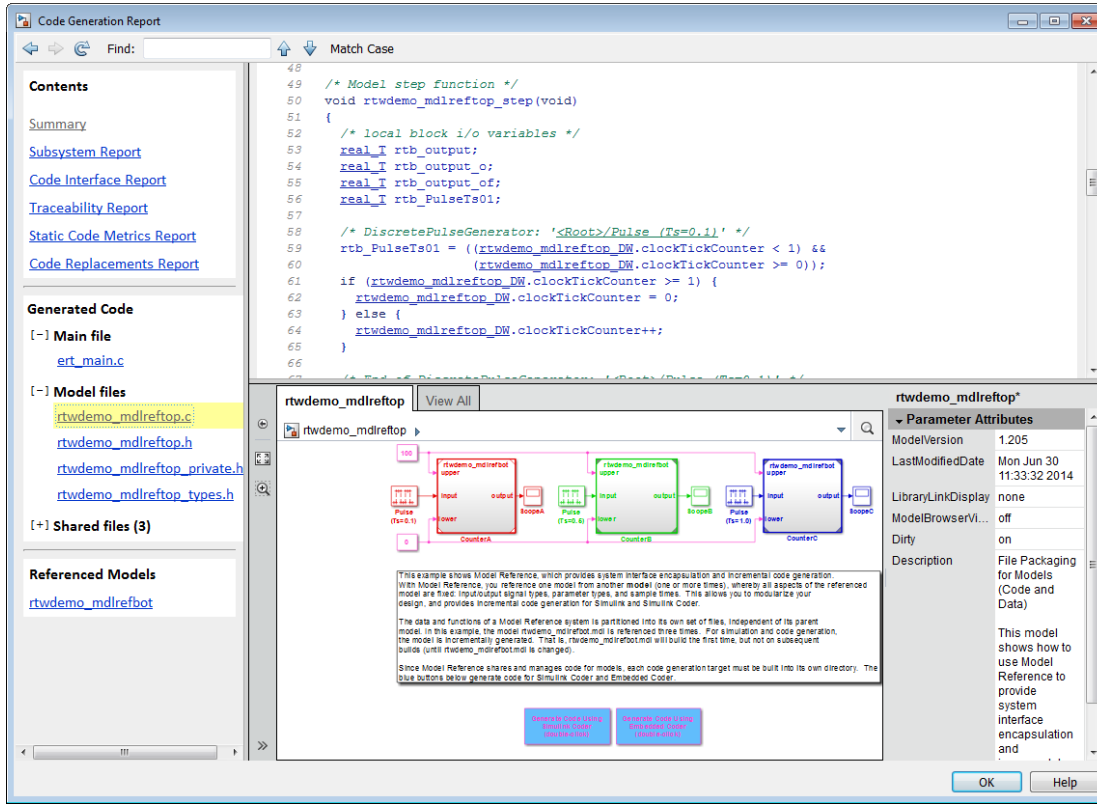
1 Open the `rtwdemo_mdlreftop` model.

2. Open the Configuration Parameters dialog box or Model Explorer and navigate to the **Code Generation** pane.
3. Specify `ert.tlc` for the **System target file** parameter.
4. Open the **Code Generation** > **Report** pane.
5. Select the following parameters:
   - **Create code generation report**
   - **Open report automatically**
   - **Generate model Web view**
6. On the **All Parameters** tab, select the parameters **Code-to-model** and **Model-to-code**.

Note: These settings specify only the top model, not referenced models.

---

7. Open the Configuration Parameters for the referenced model, `rtwdemo_mdlrefbot` and perform steps 3–6.
8. Save the models, `rtwdemo_mdlreftop` and `rtwdemo_mdlrefbot`.
9. From the top model diagram, press Ctrl+B. After building the model and generating code, the code generation report for the top model opens in a MATLAB Web browser.
10. In the left navigation pane, select a source code file. The corresponding source code is displayed in the right pane and includes hyperlinks.



11 Click a link in the code. The model Web view displays and highlights the corresponding block in the model.

12 To highlight the generated code for a referenced model block in your model, click `CounterB`. The corresponding code is highlighted in the source code pane.

Note: You cannot open the referenced model diagram in the Web view by double-clicking the referenced model block in the top model.

13 To open the code generation report for a referenced model, in the left navigation pane, below **Referenced Models**, click the link, `rtwdemo_mdlrefbot`. The source files for the referenced model are displayed along with the Web view of the referenced model.

14 To go back to the code generation report for the top model, at the top of the left navigation pane, click the **Back** button until the top model's report is displayed.

For more information about exploring a model in a Web view, see "Navigate the Web View" (Simulink Report Generator).

For more information about navigating between the generated code and the model diagram, see :

- "Trace Model Objects to Generated Code" on page 61-8
- "Trace Code to Model Objects by Using Hyperlinks" on page 61-6

Model Web View Limitations

The HTML code generation report includes the following limitations when using the model Web view:

- Code is not generated for virtual blocks. In the model Web view of the code generation report, when tracing between the model and the code, when you click a virtual block, it is highlighted yellow.
- In the model Web view, you cannot open a referenced model diagram by double-clicking the referenced model block in the top model. Instead, open the code generation report for the referenced model by clicking a link under **Referenced Models** in the left navigation pane.
- Stateflow truth tables, events, and links to library charts are not supported in the model Web view.
- Searching in the code generation report does not find or highlight text in the model Web view.
- If you navigate from the actual model diagram (not the model Web view in the report), to the source code in the HTML code generation report, the model Web view is disabled and not visible. To enable the model Web view, open the report again, see "Open Code Generation Report" (Simulink Coder).
- For a subsystem build, the traceability hyperlinks of the root level inport and outport blocks are disabled.
- "Traceability Limitations" on page 61-32 that apply to tracing between the code and the actual model diagram.

Analyze the Generated Code Interface

In this section...

Code Interface Report Overview

When you select the **Create code generation report** option for an ERT-based model, a **Code Interface Report** section is automatically included in the generated HTML report. The **Code Interface Report** section provides documentation of the generated code interface, including model entry-point functions and interface data, for consumers of the generated code. The information in the report can help facilitate code review and code integration.

The code interface report includes the following subsections:

- **Entry point functions** — interface information about each model entry-point function, including `model_initialize`, `model_step`, and (if applicable) `model_reset` and `model_terminate`.
- **Inports** and **Outports** — interface information about each model inport and outport.
- **Interface Parameters** — interface information about tunable parameters that are associated with the model.
- **Data Stores** — interface information about global data stores and data stores with non-`auto` storage that are associated with the model.

For limitations that apply to code interface reports, see "Code Interface Report Limitations" on page 35-32.

For illustration purposes, this section uses the following models:

- `rtwdemo_basicsc` (with the **ExportedGlobal Storage Class** button selected in the model window) for examples of report subsections
- `rtwdemo_mrmtbb` for examples of timing information
- `rtwdemo_fcnprotoctrl` for examples of function argument and return value information

Generating a Code Interface Report

To generate a code interface report for your model:

1. Open your model, go to the **Code Generation** pane of the Configuration Parameters dialog box, and select `ert.tlc` or an ERT-based **System target file**, if one is not already selected.
2. Go to the **Code Generation** > **Report** pane of the Configuration Parameters dialog box and select the option **Create code generation report**, if it is not already selected. The `rtwdemo_basicsc`, `rtwdemo_mrmtbb`, and `rtwdemo_fcnprotoctrl` models used in this section select multiple **Report** pane options by default. But selecting only **Create code generation report**, generates a **Code Interface Report** section in the HTML report.

   Alternatively, you can programmatically select the option by issuing the following MATLAB command:

   ```
   set_param(bdroot, 'GenerateReport', 'on')
   ```

   If the **All parameters** tab option **Code-to-model** is selected, the generated report contains hyperlinks to the model. Leave this value selected unless you plan to use the report outside the MATLAB environment.
3. Build the model. If you selected the **Report** pane option **Open report automatically**, the code generation report opens automatically after the build process is complete. (Otherwise, you can open it manually from within the model build folder.)
4. To display the code interface report for your model, go to the **Contents** pane of the HTML report and click the **Code Interface Report** link. For example, here is the generated code interface report for the model `rtwdemo_basicsc` (with the **ExportedGlobal Storage Class** button selected in the model window).

Code Interface Report for rtwdemo_basicsc

Table of Contents

Entry Point Functions

Function: rtwdemo_basicsc_initialize

Prototype	**void rtwdemo_basicsc_initialize(void)**
Description	Initialization entry point of generated code
Timing	Must be called exactly once
Arguments	None
Return value	None
Header file	rtwdemo_basicsc.h

Function: rtwdemo_basicsc_step

Prototype	**void rtwdemo_basicsc_step(void)**
Description	Output entry point of generated code
Timing	Must be called periodically, every 1 second
Arguments	None
Return value	None
Header file	rtwdemo_basicsc.h

Inports

[-]

Block Name	Code Identifier	Data Type	Dimension
<Root>/In1	input1	real32_T	1
<Root>/In2	input2	real32_T	1
<Root>/In3	input3	real32_T	1
<Root>/In4	input4	real32_T	1

Outports

Block Name	Code Identifier	Data Type	Dimension
<Root>/Out1	output	real32_T	1

Interface Parameters

[-]

Parameter Source	Code Identifier	Data Type	Dimension
K2	K2	real_T	1
LOWER	LOWER	real32_T	1
T1Break	T1Break	real32_T	[1 11]
T1Data	T1Data	real32_T	[1 11]
T2Break	T2Break	real32_T	[1 3]
T2Data	T2Data	real32_T	[3 3]
UPPER	UPPER	real32_T	1
K1	K1	int8_T	1

Data Stores

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

For help navigating the content of the code interface report subsections, see "Navigating Code Interface Report Subsections" on page 35-24. For help interpreting the content of the code interface report subsections, see the sections beginning with "Interpreting the Entry Point Functions Subsection" on page 35-25.

Navigating Code Interface Report Subsections

To help you navigate code interface descriptions, the code interface report provides collapse/expand tokens and hyperlinks, as follows:

- For a large subsection, the report provides `[-]` and `[+]` symbols that allow you to collapse or expand that section. In the example in the previous section, the symbols are provided for the **Inports** and **Interface Parameters** sections.
- Several forms of hyperlink navigation are provided in the code interface report. For example:
  - The **Table of Contents** located at the top of the code interface report provides links to each subsection.
  - You can click each function name to go to its definition in *`model`*`.c`.
  - You can click each function's header file name to go to the header file source listing.
  - If you selected the **All Parameters** tab option **Code-to-model** for your model, to go to the corresponding location in the model display, you can click hyperlinks for any of the following:
    - Function argument
    - Function return value
    - Inport
    - Outport
    - Interface parameter (if the parameter source is a block)
    - Data store (if the data store source is a Data Store Memory block)

For backward and forward navigation within the HTML code generation report, use the **Back** and **Forward** buttons above the **Contents** section in the upper-left corner of the report.

Interpreting the Entry Point Functions Subsection

The **Entry Point Functions** subsection of the code interface report provides the following interface information about each model entry-point function, including `model_initialize`, `model_step`, and (if applicable) `model_reset` and `model_terminate`.

Field	Description
Function:	Lists the function name. You can click the function name to go to its definition in *`model`*`.c`.
Prototype	Displays the function prototype, including the function return value, name, and arguments.
Description	Provides a text description of the function's purpose in the application.
Timing	Describes the timing characteristics of the function, such as how many times the function is called, or if it is called periodically, and at what time interval. For a multirate timing example, see the following `rtwdemo_mrmtbb` report excerpt.
Arguments	If the function has arguments, displays the number, name, data type, and Simulink description for each argument. If you select the **All Parameters** tab option **Code-to-model** for your model, you can click the hyperlink in the description to go to the block corresponding to the argument in the model display. For argument examples, see the `rtwdemo_fcnprotoctrl` report excerpt below.
Return value	If the function has a return value, this field displays the return value data type and Simulink description. If you selected the **All Parameters** tab option **Code-to-model** for your model, you can click the hyperlink in the description to go to the block corresponding to the return value in the model display. For a return value example, see the following `rtwdemo_fcnprotoctrl` report excerpt.
Header file	Lists the name of the header file for the function. You can click the header file name to go to the header file source listing.

For example, here is the **Entry Point Functions** subsection for the model `rtwdemo_basicsc`.

Entry Point Functions

Function: rtwdemo_basicsc_initialize

Prototype	**void rtwdemo_basicsc_initialize(void)**
Description	Initialization entry point of generated code
Timing	Must be called exactly once
Arguments	None
Return value	None
Header file	rtwdemo_basicsc.h

Function: rtwdemo_basicsc_step

Prototype	**void rtwdemo_basicsc_step(void)**
Description	Output entry point of generated code
Timing	Must be called periodically, every 1 second
Arguments	None
Return value	None
Header file	rtwdemo_basicsc.h

To illustrate how timing information might be listed for a multirate model, here are the **Entry Point Functions** and **Inports** subsections for the model `rtwdemo_mrmtbb`. This multirate, discrete-time, multitasking model contains Inport blocks 1 and 2, which specify 1-second and 2-second sample times, respectively. The sample times are constrained to the specified times by the **Periodic sample time constraint** option on the **Solver** pane of the Configuration Parameters dialog box.

Entry Point Functions

Function: rtwdemo_mrmtbb_initialize

Prototype	**void rtwdemo_mrmtbb_initialize(void)**
Description	Initialization entry point of generated code
Timing	Must be called exactly once
Arguments	None
Return value	None
Header file	rtwdemo_mrmtbb.h

Function: rtwdemo_mrmtbb_step0

Prototype	**void rtwdemo_mrmtbb_step0(void)**
Description	Output entry point of generated code
Timing	Must be called periodically, every 1 second
Arguments	None
Return value	None
Header file	rtwdemo_mrmtbb.h

Function: rtwdemo_mrmtbb_step1

Prototype	**void rtwdemo_mrmtbb_step1(void)**
Description	Output entry point of generated code
Timing	Must be called periodically, every 2 seconds
Arguments	None
Return value	None
Header file	rtwdemo_mrmtbb.h

Function: rtwdemo_mrmtbb_terminate

Prototype	**void rtwdemo_mrmtbb_terminate(void)**
Description	Termination entry point of generated code
Timing	Must be called exactly once
Arguments	None
Return value	None
Header file	rtwdemo_mrmtbb.h

Inports

Block Name	Code Identifier	Data Type	Dimension
<Root>/In1_1s	rtwdemo_mrmtbb_U.In1_1s	real_T	1
<Root>/In2_2s	rtwdemo_mrmtbb_U.In2_2s	real_T	1

To illustrate how function arguments and return values are displayed in the report, here is the entry-point function description of the model step function for model `rtwdemo_fcnprotoctrl`.

Function: rtwdemo_fcnprotoctrl_step_custom

Prototype	**boolean_T rtwdemo_fcnprotoctrl_step_custom(const real_T argIn1, const BusObject *const argIn2, BusObject *argOut2, const BusObject *const argIn3, uint8_T *argIn4)**
Description	Output entry point of generated code
Timing	Can be called at any time
Arguments	[-]

#	Name	Data Type	Description
1	argIn1	const real_T	<Root>/In1
2	argIn2	const BusObject *const	<Root>/In2
3	argOut2	BusObject *	<Root>/Out2
4	argIn3	const BusObject *const	<Root>/In3
5	argIn4	uint8_T *	<Root>/In4

Return value	Data Type	Description
	boolean_T	<Root>/Out1
Header file	rtwdemo_fcnprotoctrl.h	

Interpreting the Inports and Outports Subsections

The **Inports** and **Outports** subsections of the code interface report provide the following interface information about each inport and outport in the model.

Field	Description
Block Name	Displays the Simulink block name of the inport or outport. If you selected the **All Parameters** tab option **Code-to-model** for your model, you can click on each inport or outport **Block Name** value to go to its location in the model display.
Code Identifier	Lists the identifier associated with the inport or outport data in the generated code, as follows: • If the data is defined in the generated code, the field displays the identifier text.

Field	Description
	• If the data is declared but not defined in the generated code — for example, if the data is resolved with an imported storage class — the field displays the identifier text prefixed with the label '`Imported data:`'. • If the data is neither defined nor declared in the generated code — for example, if **Reusable function** code interface packaging is selected for the model — the field displays the text '`Defined externally`'.
Data Type	Lists the data type of the inport or outport.
Scaling	For fixed-point entries, lists the data type and fraction length using Simulink fixed-point data type notation. **Note:** You must have a Fixed-Point Designer license to see fixed-point scaling information in the report. For more information on how scaling is represented in the table, see "Fixed-Point Data Type and Scaling Notation" (Fixed-Point Designer).
Dimension	Lists the dimensions of the inport or outport (for example, `1` or `[4, 5]`).

For example, here are the **Inports** and **Outports** subsections for the model `rtwdemo_basicsc`.

Inports

[-]

Block Name	Code Identifier	Data Type	Dimension
<Root>/In1	input1	real32_T	1
<Root>/In2	input2	real32_T	1
<Root>/In3	input3	real32_T	1
<Root>/In4	input4	real32_T	1

Outports

Block Name	Code Identifier	Data Type	Dimension
<Root>/Out1	output	real32_T	1

Interpreting the Interface Parameters Subsection

The **Interface Parameters** subsection of the code interface report provides the following interface information about tunable parameters that are associated with the model.

Field	Description
Parameter Source	Lists the source of the parameter value, as follows: • If the source of the parameter value is a block, the field displays the block name, such as `<Root>/Gain2` or `<S1>/Lookup1`. If you selected the **All Parameters** tab option **Code-to-model** for your model, you can click the **Parameter Source** value to go to the parameter's location in the model display. • If the source of the parameter value is a workspace variable, the field displays the name of the workspace variable.
Code Identifier	Lists the identifier associated with the tunable parameter data in the generated code, as follows: • If the data is defined in the generated code, the field displays the identifier text. • If the data is declared but not defined in the generated code — for example, if the data is resolved with an imported storage class — the field displays the identifier text prefixed with the label `'Imported data:'`. • If the data is neither defined nor declared in the generated code — for example, if **Reusable function** code interface packaging is selected for the model — the field displays the text `'Defined externally'`.
Data Type	Lists the data type of the tunable parameter.
Scaling	For fixed-point entries, lists the data type and fraction length using Simulink fixed-point data type notation. **Note:** You must have a Fixed-Point Designer license to see fixed-point scaling information in the report. For more information on how scaling is represented in the table, see "Fixed-Point Data Type and Scaling Notation" (Fixed-Point Designer).

Field	Description
Dimension	Lists the dimensions of the tunable parameter (for example, `1` or `[4, 5, 6]`).

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

Interface Parameters

[-]

Parameter Source	Code Identifier	Data Type	Dimension
K2	K2	real_T	1
LOWER	LOWER	real32_T	1
T1Break	T1Break	real32_T	[1 11]
T1Data	T1Data	real32_T	[1 11]
T2Break	T2Break	real32_T	[1 3]
T2Data	T2Data	real32_T	[3 3]
UPPER	UPPER	real32_T	1
K1	K1	int8_T	1

Interpreting the Data Stores Subsection

The **Data Stores** subsection of the code interface report provides the following interface information about global data stores and data stores with non-`auto` storage that are associated with the model.

Field	Description
Data Store Source	Lists the source of the data store memory, as follows: • If the data store is defined using a Data Store Memory block, the field displays the block name, such as `<Root>/DS1`. If you selected the **All Parameters** tab option **Code-to-model** for your model, you can click on the **Data Store Source** value to go to the data store's location in the model display. • If the data store is defined using a `Simulink.Signal` object, the field displays the name of the `Simulink.Signal` object.
Code Identifier	Lists the identifier associated with the data store data in the generated code, as follows: • If the data is defined in the generated code, the field displays the identifier text.

Field	Description
	• If the data is declared but not defined in the generated code — for example, if the data is resolved with an imported storage class — the field displays the identifier text prefixed with the label '`Imported data:`'. • If the data is neither defined nor declared in the generated code — for example, if **Reusable function** code interface packaging is selected for the model — the field displays the text '`Defined externally`'.
Data Type	Lists the data type of the data store.
Scaling	For fixed-point entries, lists the data type and fraction length using Simulink fixed-point data type notation. **Note:** You must have a Fixed-Point Designer license to see fixed-point scaling information in the report. For more information on how scaling is represented in the table, see "Fixed-Point Data Type and Scaling Notation" (Fixed-Point Designer).
Dimension	Lists the dimensions of the data store (for example, `1` or `[1, 2]`).

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

Data Stores

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

Code Interface Report Limitations

The following limitations apply to the code interface section of the HTML code generation reports.

- The code interface report does not support the GRT interface with an ERT target or **C ++ class** code interface packaging. For these configurations, the code interface report is not generated and does not appear in the HTML code generation report **Contents** pane.
- The code interface report supports data resolved with most custom storage classes (CSCs), except when the CSC properties are set in any of the following ways:

  - The CSC property **Type** is set to `FlatStructure`. For example, the `BitField` and `Struct` CSCs in the Simulink package have **Type** set to `FlatStructure`.
  - The CSC property **Type** is set to `Other`. For example, the `GetSet` CSC in the Simulink package has **Type** set to `Other`.
  - The CSC property **Data access** is set to `Pointer`, indicating that imported symbols are declared as pointer variables rather than simple variables. This property is accessible only when the CSC property **Data scope** is set to `Imported` or `Instance-specific`.

  In these cases, the report displays empty **Data Type** and **Dimension** fields.
- For outports, the code interface report cannot describe the associated memory (data type and dimensions) if the memory is optimized. In these cases, the report displays empty **Data Type** and **Dimension** fields.
- The code interface report does not support data type replacement using the **Code Generation > Data Type Replacement** pane of the Configuration Parameters dialog box. The data types listed in the report will link to built-in data types rather than their specified replacement data types.

Related Examples

- “Design Data Interface by Configuring Inport and Outport Blocks” on page 19-134

Static Code Metrics

In this section...

About Static Code Metrics

The code generator performs static analysis of the generated C or C++ code and provides these metrics in the **Static Code Metrics Report** section of the HTML Code Generation Report.

You can use the information in the report to:

- Find the number of files and lines of code in each file.
- Estimate the number of lines of code and stack usage per function.
- Compare the difference in terms of how many files, functions, variables, and lines of code are generated every time you change the model or MATLAB algorithm.
- Determine a target platform and allocation of RAM to the stack, based on the size of global variables plus the estimated stack size.
- Determine possible performance slow points, such as the largest global variables or the most costly call path in terms of stack usage.
- View the cyclomatic complexity of a function, which counts the number of linearly independent paths through a function.
- View the function call tree. Determine the longest call path to estimate the worst case execution timing.
- View how target functions, provided by the selected code replacement library, are used in the generated code.

For examples, see

- "Generate Static Code Metrics Report for Simulink Model" on page 35-38
- "Generate a Static Code Metrics Report for MATLAB Code" on page 35-43

Static Code Metrics Analysis

Static analysis of the generated code is performed only on the source code without executing the program. The results of the static code metrics analysis are included in the **Static Code Metrics** section of the HTML Code Generation Report. The report is not available if you generate a MEX function from MATLAB code.

Static analysis of the generated source code files:

- Uses the specified C data types. For Simulink models, you specify these data types in the **Hardware Implementation** > **Production hardware** pane of the Configuration Parameters dialog box. For code generation from MATLAB code, you specify them in the **Hardware** tab of the MATLAB Coder project settings dialog box or using a code generation configuration object. Actual object code metrics might differ due to target-specific compiler and platform settings.
- Includes custom code only if you specify it. For Simulink models, you specify custom code on the **Code Generation** > **Custom Code** pane in the model configuration. For code generation from MATLAB code, you specify it on the **Debugging** tab of the MATLAB Coder project settings dialog box or using a code generation configuration object. An error report is generated if the generated code includes platform-specific files not contained in the standard C run-time library.
- For Simulink models, includes the generated code from referenced models.
- Uses 1-byte alignment for all members of a structure for estimating global and local data structure sizes. The size of a structure is calculated by summing the sizes of all of its fields. This estimation represents the smallest possible size for a structure.
- Calculates the self stack size of a function as the size of local data within a function, excluding input arguments. The accumulated stack size of a function is the self stack size plus the maximum of the accumulated stack sizes of its called functions. For example, if the accumulated stacks sizes for the called functions are represented as `accum_size1...accum_sizeN`, then the accumulated stack size for a function is

  ```
  accumulated_stack_size = self_stack_size + max(accum_size1,...,accum_sizeN)
  ```
- When estimating the stack size of a function, static analysis stops at the first instance of a recursive call. The **Function Information** table indicates when recursion occurs in a function call path. Code generation generates only recursive code for Stateflow event broadcasting and for graphical functions if it is written as a recursive function.
- Calculates the cyclomatic complexity of a function as the number of decisions plus one:

  ```
  CC = Number of decisions + 1
  ```

The following constructs add a decision:

- If statement
- Else-If statement
- Switch statement (1 decision for each `case` branch)
- Loop statements: While, For, Do-while

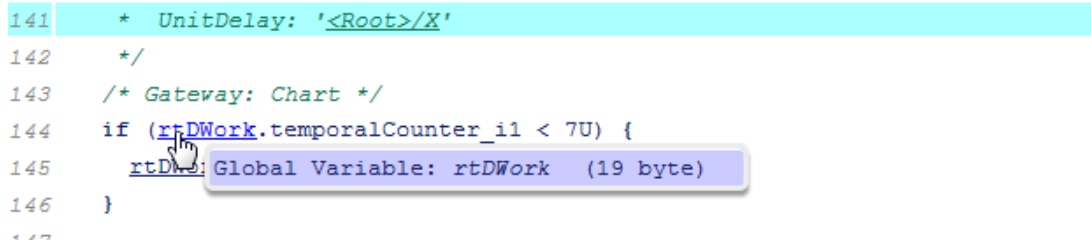
Note: Boolean operators in the above constructs do not add extra decisions.

- Does not include `ert_main.c`, because you have the option to provide your own `main.c`.

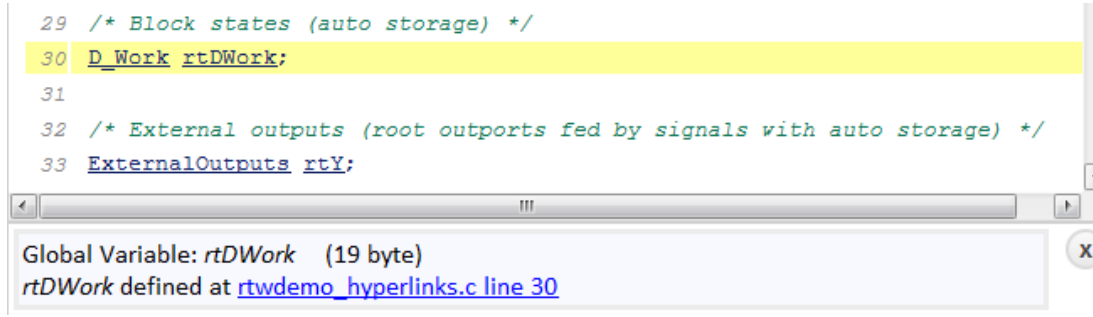
View Static Code Metrics and Definitions Within the Generated Code

When you view code in the code generation report, to get access to code metrics and definitions, you can use the following tools:

- On the **Code Generation > Report** pane, if you select the **Static code metrics** check box you can hover your cursor over global variables and functions in the code window to see code metrics information.



- In the code window, if you click linked variables or functions, the code inspect window is displayed. The window provides links to definitions for the variables or functions. On the **Code Generation > Report** pane, if you selected the **Static code metrics** check box, you can also see code metrics information for the variable or function.


29 /* Block states (auto storage) */
30 D_Work rtDWork;
31
32 /* External outputs (root outports fed by signals with auto storage) */
33 ExternalOutputs rtY;
Global Variable: rtDWork (19 byte)
rtDWork defined at rtwdemo_hyperlinks.c line 30

Generate Static Code Metrics Report for Simulink Model

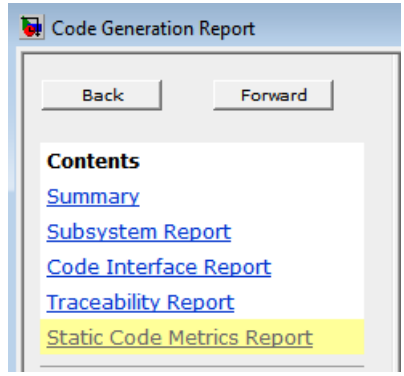
The **Static Code Metrics Report** is a section included in the HTML Code Generation Report. For more information on the static analysis of the generated code, see "Static Code Metrics Analysis" on page 35-35.

1. Before generating the HTML Code Generation Report, open the Configuration Parameters dialog box for your model. On the **Code Generation > Report** pane, select the "Static code metrics" (Simulink Coder) check box.

   If your model includes referenced models, select the **Static code metrics** check box in each referenced model's configuration set. Otherwise, you cannot view a separate static code metrics report for a referenced model.

2. Press **Ctrl+B** to build your model and generate the HTML code generation report. For more information, see "Traceability in Code Generation Report" on page 35-15.

3. If the HTML Code Generation Report is not already open, open the report. On the left navigation pane, in the **Contents** section, select **Static Code Metrics Report**.



4. To see the generated files and how many lines of code are generated per file, look at the **File Information** section.

1. File Information [hide]

[-] Summary (excludes ert_main.c)

Number of .c files : 2
Number of .h files : 4
Lines of code : 1,132
Lines : 1,958

[-] File details

File Name	Lines of Code	Lines	Generated On
fuel_rate_control.c	665	1,107	06/10/2016 4:17 PM
fuel_rate_control_data.c	253	342	06/10/2016 4:17 PM
rtwtypes.h	97	177	06/10/2016 4:17 PM
fuel_rate_control.h	88	249	06/10/2016 4:17 PM
fuel_rate_control_types.h	22	52	06/10/2016 4:17 PM
fuel_rate_control_private.h	7	31	06/10/2016 4:17 PM

5 Hover your cursor over column titles and some column values to see a description of the corresponding data.

6 If your model includes referenced models, the File information section includes a **Referenced Model** column. In this column, click the referenced model name to open its static code metrics report. If the static code metrics report is not available for a referenced model, specify the **Static code metrics** parameter in the referenced model's configuration set and rebuild your model.

7 To view the global variables in the generated code, their size, and the number of accesses, see the **Global Variables** section.

2. Global Variables [hide]

Global variables defined in the generated code.

Global Variable	Size (bytes)	Reads / Writes	Reads / Writes in a Function
[+] fuel_rate_control_DW	23	175	78
[+] fuel_rate_control_B	21	42	26
[+] fuel_rate_control_U	16	27	24
[+] fuel_rate_control_M_	8	0*	0*
[+] fuel_rate_control_Y	4	4	3
Total	**72**	**248**	

* The global variable is not directly used in any function.

The **Reads/Writes** column displays the total number of read and write accesses to the global variable. The **Reads/Writes in a Function** column displays the maximum number of read and write accesses to the global variable within a function. You use this information is to estimate the benefit of turning on optimizations, which

reduce the number of global references. For more information, see “Optimize Global Variable Usage” on page 55-2.

Click [+] to expand structures.

Global variables defined in the generated code.

Global Variable	Size (bytes)	Reads / Writes	Reads / Writes in a Function
[−] fuel_rate_control_DW	23	175	78
DiscreteIntegrator_DSTATE	4	5	4
ThrottleTransient_states	4	4	3
DiscreteFilter_states	4	4	3
DiscreteFilter_states_i	4	4	3
[+] bitsForTID0	5	156	60
temporalCounter_i1	2	7	5
[−] fuel_rate_control_B	21	42	26
[+] es_o	16	23	18
fuel_mode	4	15	8
O2_normal	1	6	4
[−] fuel_rate_control_U	16	27	24
[+] sensors	16	27	24
[−] fuel_rate_control_M_	8	0*	0*
errorStatus	8	0	0
[−] fuel_rate_control_Y	4	4	3
fuel_rate	4	4	3
Total	**72**	**248**	

* The global variable is not directly used in any function.

8 To navigate from the report to the source code, click a global variable or function name. These names are hyperlinks to their definitions.

9 To view the function call tree of the generated code, in the **Function Information** section, click **Call Tree** at the top of the table.

3. Function Information [hide]

View function metrics in a call tree format or table format. Accumulated stack numbers include the estimated stack size of the function plus the maximum of the accumulated stack size of the subroutines that the function calls.

View:Call Tree | Table

Function Name	Accumulated Stack Size (bytes)	Self Stack Size (bytes)	Lines of Code	Lines	Complexity
[-] fuel_rate_control_step	52	16	272	504	43
look2_iflf_linlca	36	36	55	100	12
[-] fuel_rate_control_Fail	4	4	67	103	13
fuel_rate_control_Fueling_Mode	0	0	166	240	19
fuel_rate_control_Fueling_Mode	0	0	166	240	19
[+] fuel_rate_control_initialize	0	0	43	67	1
fuel_rate_control_terminate	0	0	0	4	1

`ert_main.c` is not included in the code metrics analysis, therefore it is not shown in the call tree format. The **Complexity** column includes the cyclomatic complexity of each function.

10 To view the functions in a table format, click **Table**.

3. Function Information [hide]

View function metrics in a call tree format or table format. Accumulated stack numbers include the estimated stack size of the function plus the maximum of the accumulated stack size of the subroutines that the function calls.

View: Call Tree | Table

Function Name	Called By (number of call sites)	Accumulated Stack Size (bytes)	Self Stack Size (bytes)	Lines of Code	Lines	Complexity
fuel_rate_control_Fail	fuel_rate_control_step (9)	4	4	67	103	13
fuel_rate_control_Fueling_Mode	fuel_rate_control_step fuel_rate_control_Fail (2)	0	0	166	240	19
fuel_rate_control_initialize		0	0	43	67	1
fuel_rate_control_step		52	16	272	504	43
fuel_rate_control_terminate		0	0	0	4	1
look2_iflf_linlca	fuel_rate_control_step (5)	36	36	55	100	12
memset	fuel_rate_control_initialize (2)	*Not available*	-	-	-	-

The second column, **Called By**, lists functions that call the function listed in the first column, using the following criteria:

- If a function is called by multiple functions, all functions are listed.
- If a function has no called function, this column is empty.

For example, `Fueling_Mode` is called by `Fail` and `fuel_rate_control_step`. The number of call sites is included in parentheses. `Fail` calls `Fueling_Mode` twice.

Generate a Static Code Metrics Report for MATLAB Code

Generate a Static Code Metrics Report Using the MATLAB Coder App

This example shows how to generate a static code metrics report for a static C library that is generated from MATLAB code using the MATLAB Coder app.

By default, if you have an Embedded Coder license, when you use MATLAB Coder to generate standalone C/C++ code, the code generation report includes a static code metrics report. The static code metrics report is not available for generated MEX functions.

Create the Example Files

1. In a local, writable folder, create a MATLAB file, `moving_average.m`, that contains:

```
function [avg,z] = moving_average(x,z)
    %#codegen
    z(2:end) = z(1:end-1);  % Update buffer
    z(1) = x;               % Add new value
    avg = mean(z);          % Compute moving average
end
```

2. In the same local, writable folder, create a test file, `moving_average_test.m`, that contains:

```
function moving_average_test( )
    z = zeros(10,1);
    for i = 1:10
        [avg, z] = moving_average(i,z);
    end
    disp(avg)
end
```

Set Up the MATLAB Coder Project

1. To open the MATLAB Coder app and set up a project, at the command line, enter:

```
coder -new moving_average.prj
```

   The app adds `moving_average` to the list of entry-point functions.

2. Click **Next** to go to the **Define Input Types** step.

Define Input Types

1. To automatically define the input types, select or enter the test file `moving_average_test.m`. Click **Autodefine Input Types**.

   The app determines that `x` is `double(1x1)` and `z` is `double(10x1)`.
2. Click **Next** to go to the **Check for Run-Times Issues** step.

   The **Check for Run-Time Issues** step generates a MEX file from your entry-point functions, runs the MEX function, and reports issues. This step is optional. However, it is a best practice to perform this step. You can detect and fix run-time errors that are harder to diagnose in the generated C code.

Check for Run-Time Issues

1. To open the **Check for Run-Time Issues** dialog box, click the **Check for Issues** arrow  .

   The app populates the test file field with `moving_average_test.m`, the test file that you used to define input types.
2. Click **Check for Issues**.

   The app does not detect issues.
3. Click **Next** to go to the **Generate Code** step.

Configure the Build Settings

1. On the **Generate Code** page, to open the **Generate** dialog box, click the **Generate** arrow  .
2. Set **Build type** to `Static library`.

   The default output file name is `moving_average`.
3. Click **More Settings**.
4. On the **Debugging** tab, verify that the **Static code metrics** check box is selected.
5. Click **Close**.

Generate C Code

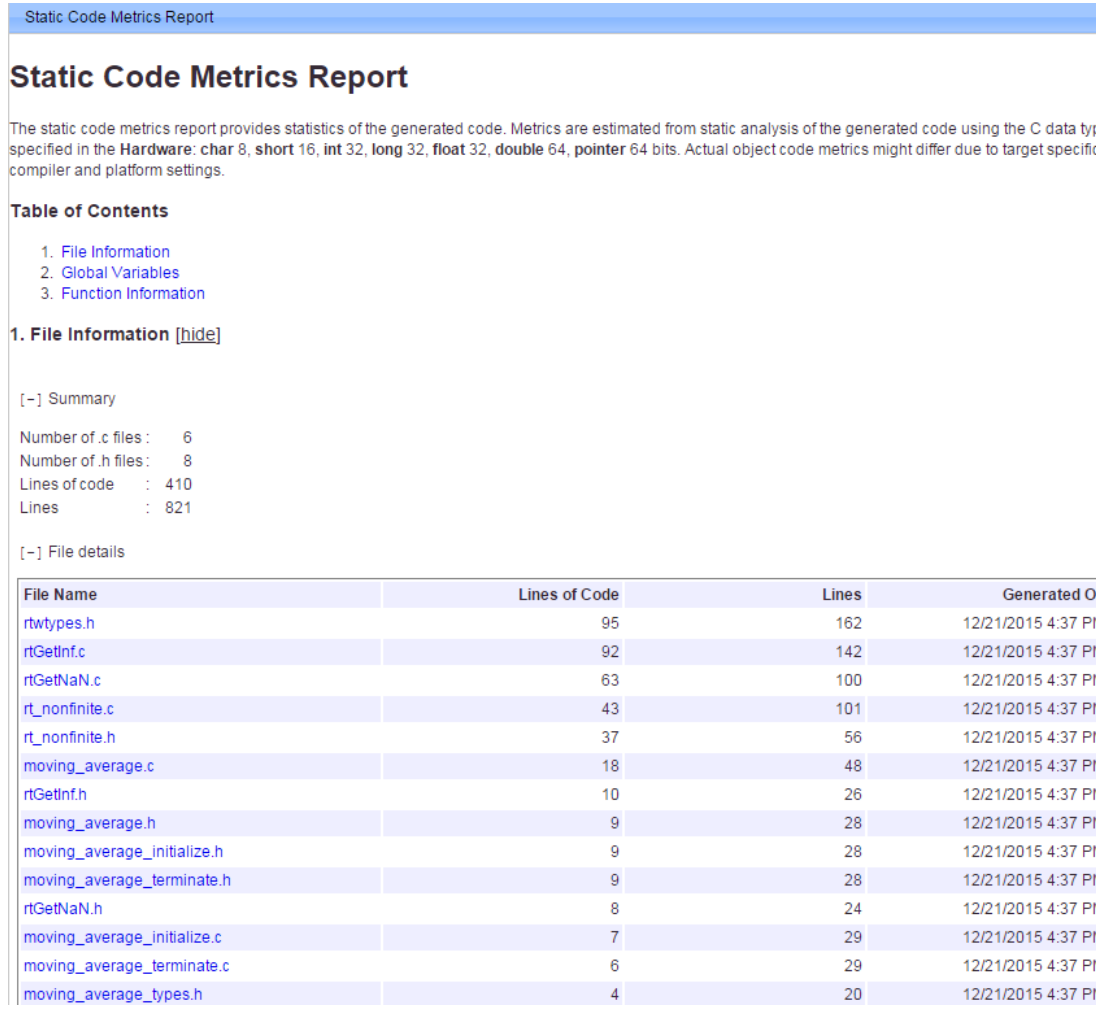
1. To generate the library, click **Generate**.

MATLAB Coder generates a C static library and supporting files in the default folder, `codegen/lib/moving_average`.

2. Click **Next** to go to the **Finish Workflow** step.

View the Static Code Metrics Report

1. To open the code generation report, under **Generated Output**, click **Code Generation Report**.
2. In the code generation report, click **Static Code Metrics Report**.
3. To see the generated files and the number of lines of code per file, click **File Information**.



Static Code Metrics Report

Static Code Metrics Report

The static code metrics report provides statistics of the generated code. Metrics are estimated from static analysis of the generated code using the C data types specified in the **Hardware**: **char** 8, **short** 16, **int** 32, **long** 32, **float** 32, **double** 64, **pointer** 64 bits. Actual object code metrics might differ due to target specific compiler and platform settings.

Table of Contents

1. File Information
2. Global Variables
3. Function Information

1. File Information [hide]

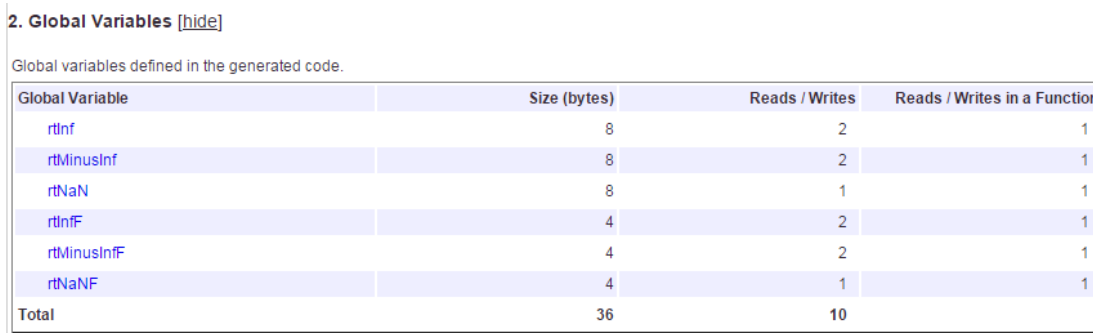
[-] Summary

Number of .c files : 6
Number of .h files : 8
Lines of code : 410
Lines : 821

[-] File details

File Name	Lines of Code	Lines	Generated On
rtwtypes.h	95	162	12/21/2015 4:37 PM
rtGetInf.c	92	142	12/21/2015 4:37 PM
rtGetNaN.c	63	100	12/21/2015 4:37 PM
rt_nonfinite.c	43	101	12/21/2015 4:37 PM
rt_nonfinite.h	37	56	12/21/2015 4:37 PM
moving_average.c	18	48	12/21/2015 4:37 PM
rtGetInf.h	10	26	12/21/2015 4:37 PM
moving_average.h	9	28	12/21/2015 4:37 PM
moving_average_initialize.h	9	28	12/21/2015 4:37 PM
moving_average_terminate.h	9	28	12/21/2015 4:37 PM
rtGetNaN.h	8	24	12/21/2015 4:37 PM
moving_average_initialize.c	7	29	12/21/2015 4:37 PM
moving_average_terminate.c	6	29	12/21/2015 4:37 PM
moving_average_types.h	4	20	12/21/2015 4:37 PM

4 To see the global variables in the generated code, go to the **Global Variables** section.



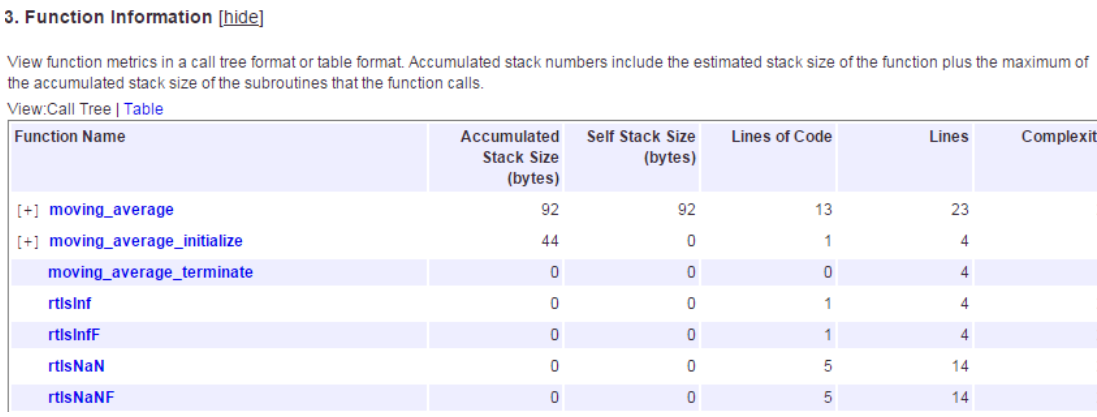
2. Global Variables [hide]

Global variables defined in the generated code.

Global Variable	Size (bytes)	Reads / Writes	Reads / Writes in a Function
rtInf	8	2	1
rtMinusInf	8	2	1
rtNaN	8	1	1
rtInfF	4	2	1
rtMinusInfF	4	2	1
rtNaNF	4	1	1
Total	36	10	

To navigate from the report to the source code, click a global variable name.

5 To view the function call tree of the generated code, in the **Function Information** section, click **Call Tree**.



3. Function Information [hide]

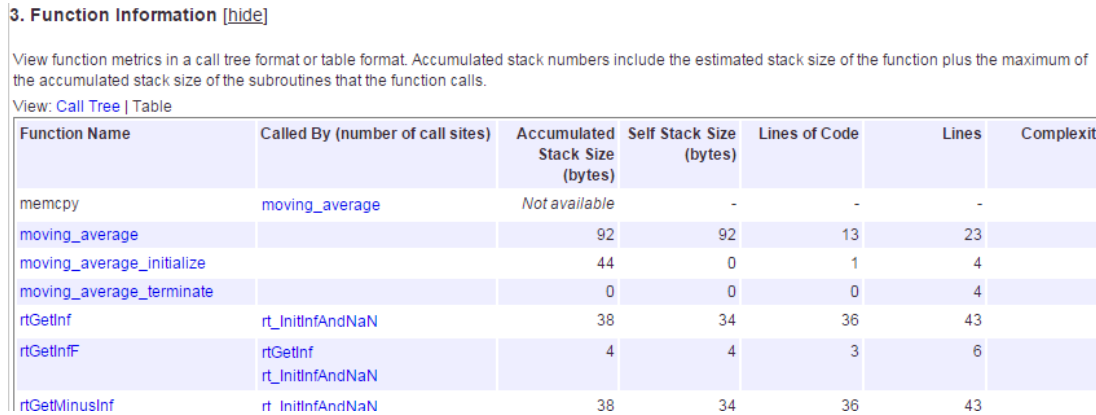
View function metrics in a call tree format or table format. Accumulated stack numbers include the estimated stack size of the function plus the maximum of the accumulated stack size of the subroutines that the function calls.

View:Call Tree | Table

Function Name	Accumulated Stack Size (bytes)	Self Stack Size (bytes)	Lines of Code	Lines	Complexity
[+] moving_average	92	92	13	23	2
[+] moving_average_initialize	44	0	1	4	1
moving_average_terminate	0	0	0	4	1
rtIsInf	0	0	1	4	2
rtIsInfF	0	0	1	4	2
rtIsNaN	0	0	5	14	2
rtIsNaNF	0	0	5	14	2

To navigate from the report to the function code, click a function name.

6 To view the functions in a table format, click **Table**.



3. Function Information [hide]

View function metrics in a call tree format or table format. Accumulated stack numbers include the estimated stack size of the function plus the maximum of the accumulated stack size of the subroutines that the function calls.

View: Call Tree | Table

Function Name	Called By (number of call sites)	Accumulated Stack Size (bytes)	Self Stack Size (bytes)	Lines of Code	Lines	Complexity
memcpy	moving_average	*Not available*	-	-	-	-
moving_average		92	92	13	23	2
moving_average_initialize		44	0	1	4	1
moving_average_terminate		0	0	0	4	1
rtGetInf	rt_InitInfAndNaN	38	34	36	43	5
rtGetInfF	rtGetInf rt_InitInfAndNaN	4	4	3	6	1
rtGetMinusInf	rt_InitInfAndNaN	38	34	36	43	5

The second column, **Called By**, lists functions that call the function listed in the first column. If multiple functions call the function, all functions are listed. If no functions call the function, this column is empty.

Enable a Static Code Metrics Report at the Command Line

To enable a static code metrics report at the command line:

1. Create a code generation configuration object for standalone code generation. For example, to generate a static library, use:

   ```
   cfg = coder.config('lib', 'ecoder', true);
   ```

2. Generate code, passing the configuration object as a parameter and specifying the `-report` option. For example:

   ```
   codegen -config cfg -report foo
   ```

Alternatively, you can:

1. Create a code generation configuration object for standalone code generation. For example, to generate a static library:

   ```
   cfg = coder.config('lib', 'ecoder', true);
   ```

2. Set the configuration object `GenerateReport` and `GenerateCodeMetricsReport` parameters to `true`.

   ```
   cfg.GenerateReport = true;
   ```

```
cfg.GenerateCodeMetricsReport = true;
```

3 Generate code, passing the configuration object as a parameter. For example:

```
codegen -config cfg foo
```

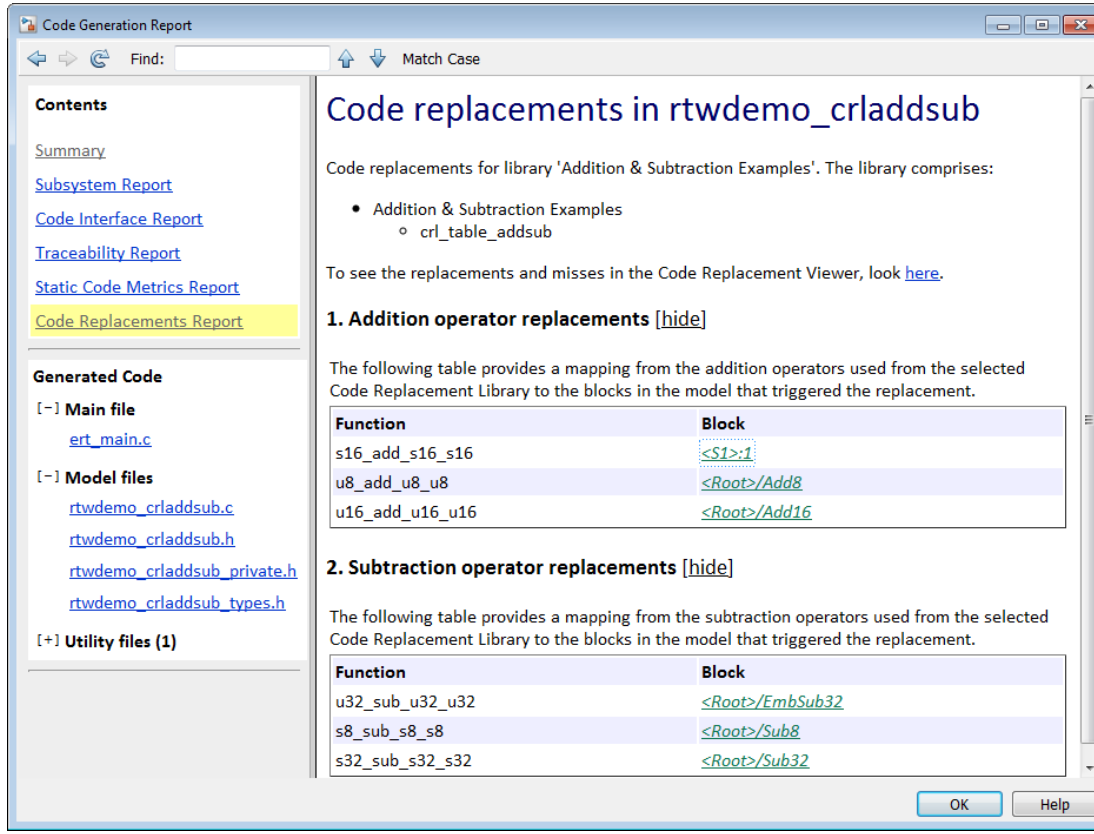
Analyze Code Replacements in Generated Code

When you select the check box **Summarize which blocks triggered code replacements** (Simulink Coder) for an ERT-based model, a Code Replacements Report section is automatically included in the generated HTML report. The Code Replacements Report section documents the code replacement library (CRL) functions that were used for code replacements during code generation, providing a mapping between each replacement instance and the Simulink block that triggered the replacement. To enable display of the Simulink block information, select the **Code Generation > Comments** check box **Include comments**. On the same pane, select either the **Simulink block / Stateflow object comments** check box or the **Simulink block descriptions** check box if present, or both.

You can use the report to:

- Determine which replacement functions were used in the generated code.
- Trace each replacement instance back to the block that triggered the replacement.

The figure below shows a Code Replacements Report generated for the CRL model rtwdemo_crladdsub. Each replacement function used is listed with a link to the block that triggered the replacement.



If you click a block path in the report, the block that triggered the replacement is highlighted in the model diagram. If the replacement was triggered by a Stateflow chart or a MATLAB function, a window opens to display the chart or function.

For more information, see Trace Code Replacements Generated Using Your Code Replacement Library on page 51-82.

Document Generated Code with Simulink Report Generator

In this section...

The Simulink Report Generator software creates documentation from your model in multiple formats, including HTML, PDF, RTF, Microsoft Word, and XML. This example shows one way to document a code generation project in Microsoft Word. The generated report includes:

- System snapshots (model and subsystem diagrams)
- Block execution order list
- Simulink Coder and model version information for generated code
- List of generated files
- Optimization configuration parameter settings
- System target file selection and build process configuration parameter settings
- Subsystem map
- File name, path, and generated code listings for the source code

To adjust Simulink Report Generator settings to include custom code and then generate a report for a model, complete the following tasks:

1. "Generate Code for the Model" on page 35-53
2. "Open the Report Generator" on page 35-53
3. "Set Report Name, Location, and Format" on page 35-55
4. "Include Models and Subsystems in a Report" on page 35-56
5. "Customize the Report" on page 35-57
6. "Generate the Report" on page 35-58

A Simulink Report Generator license is required for the following report formats: PDF, RTF, Microsoft Word, and XML. For more information on generating reports in these formats, see the Simulink Report Generator documentation.

Generate Code for the Model

Before you use the Report Generator to document your project, generate code for the model.

1. In the MATLAB Current Folder browser, navigate to a folder where you have write access.
2. Create a working folder from the MATLAB command line by typing:

   ```
   mkdir report_ex
   ```

3. Make `report_ex` your working folder:

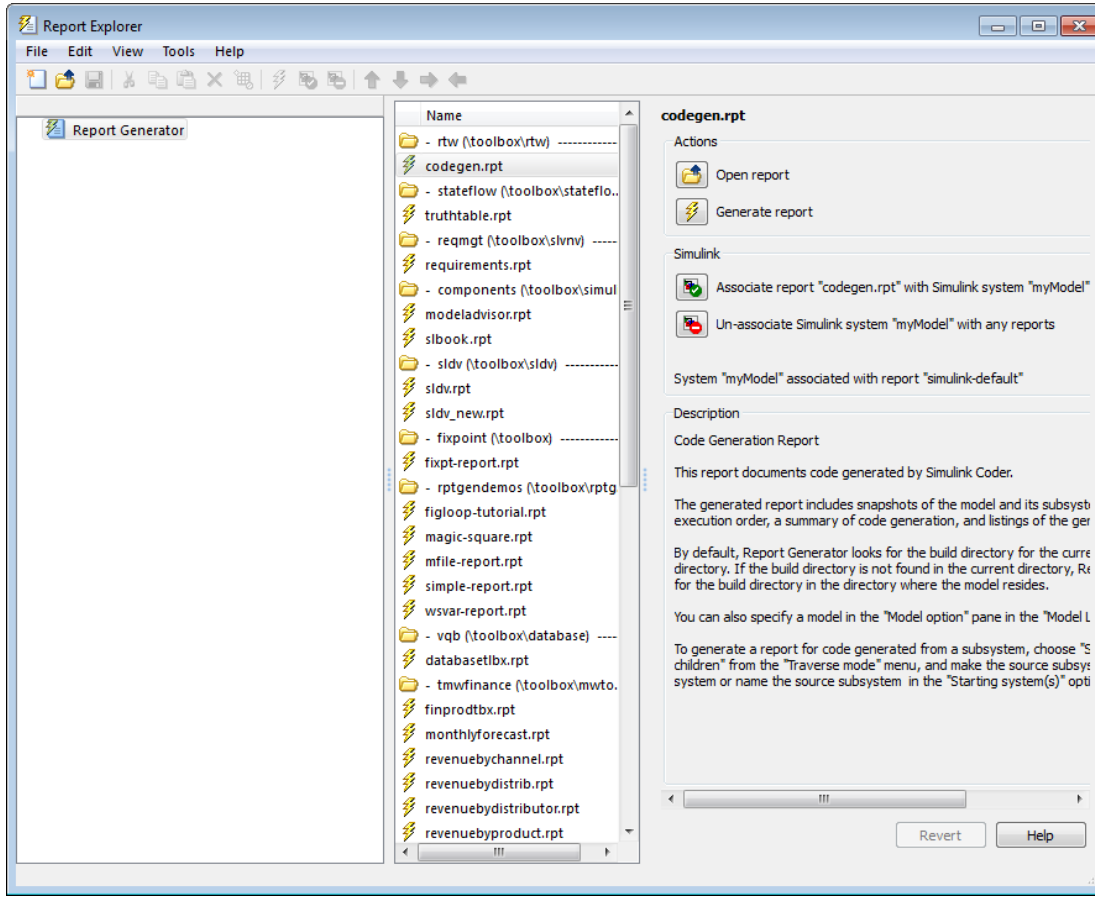
   ```
   cd report_ex
   ```

4. Open the `slexAircraftExample` model by entering the model name on the MATLAB command line.
5. In the model window, choose **File > Save As**, navigate to the working folder, `report_ex`, and save a copy of the `slexAircraftExample` model as `myModel`.
6. Open the Configuration Parameters dialog box by selecting **Model Configuration Parameters** from the **Simulation** menu.
7. Select the **Solver** pane. In the **Solver options** section, specify the **Type** parameter as `Fixed-step`.
8. Select the **Code Generation** pane. Select **Generate code only**.
9. Click **Apply**.
10. In the model window, press **Ctrl+B**. The build process generates code for the model.

Open the Report Generator

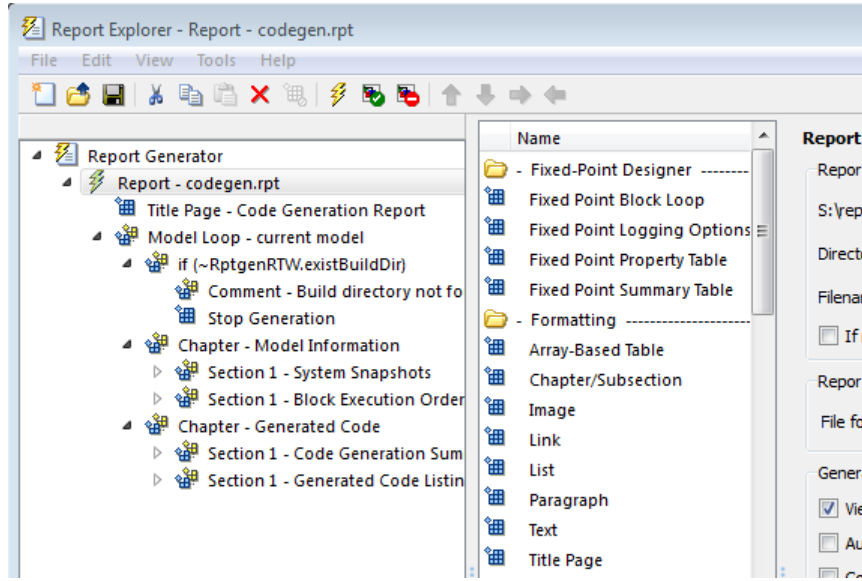
After you generate the code, open the Report Generator.

1. In the model diagram window, select **Tools > Report Generator**.
2. In the Report Explorer window, in the options pane (center), click the folder **rtw (\toolbox\rtw)**. Click the setup file that it contains, **codegen.rpt**.



3

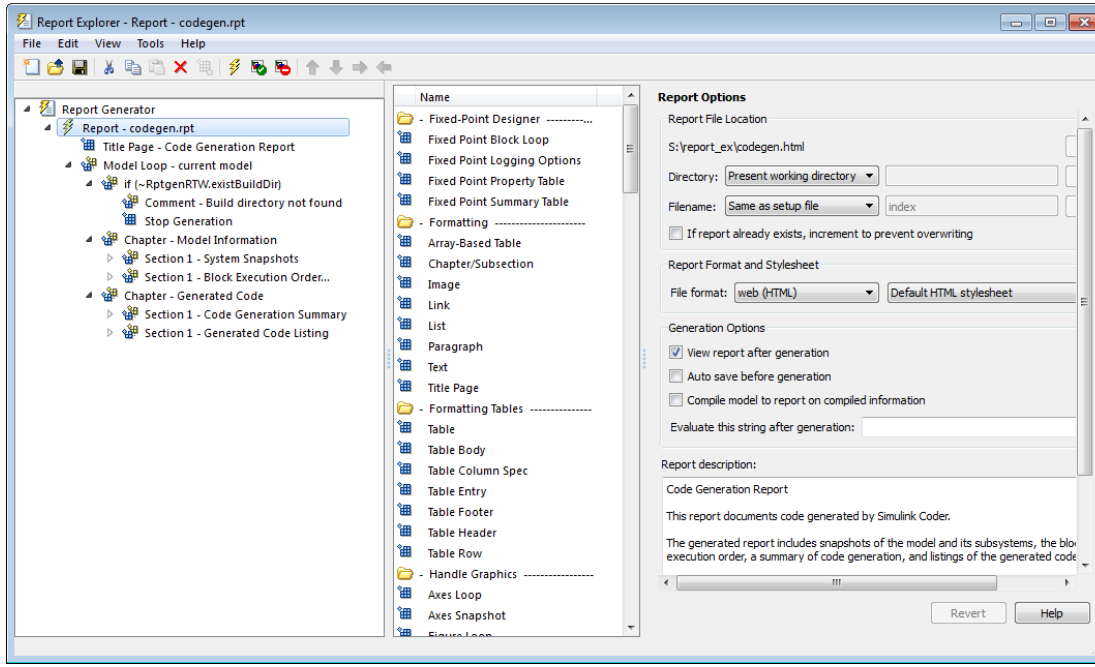
Double-click **codegen.rpt** or select it and click the **Open report** button  . The Report Explorer displays the structure of the setup file in the outline pane (left).



Set Report Name, Location, and Format

Before generating a report, you can specify report output options, such as the folder, file name, and format. For example, to generate a Microsoft Word report named `MyCGModelReport.rtf`:

1 In the properties pane, under **Report Options**, review the options listed.



2 Leave the **Directory** field set to `Present working directory`.

3 For **Filename**, select `Custom:` and replace `index` with the name `MyModelCGReport`.

4 For **File format**, specify `Rich Text Format` and replace `Standard Print` with `Numbered Chapters & Sections`.

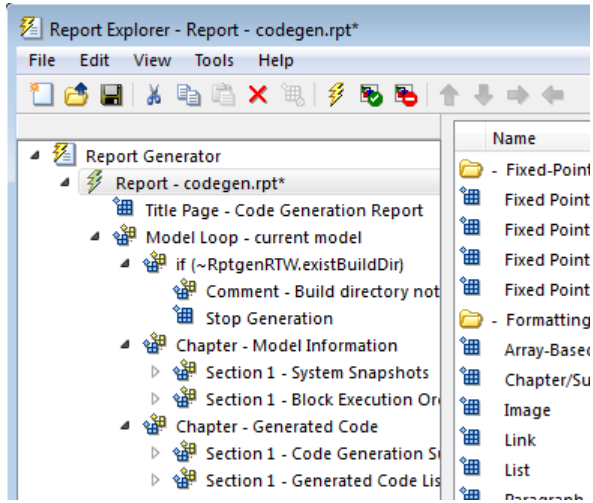
Include Models and Subsystems in a Report

Specify the models and subsystems that you want to include in the generated report by setting options in the Model Loop component.

1 In the outline pane (left), select **Model Loop**. Report Generator displays Model Loop component options in the properties pane.

2 If not already selected, select `Current block diagram` for the **Model name** option.

3 In the outline pane, click **Report - codegen.rpt***.



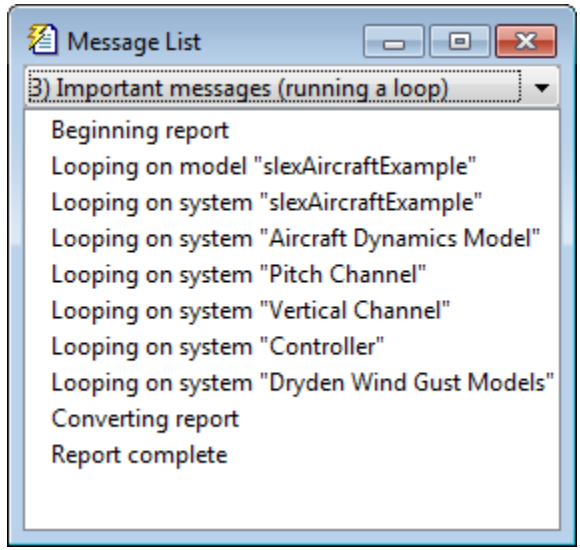
Customize the Report

After specifying the models and subsystems to include in the report, you can customize the sections included in the report.

1. In the outline pane (left), expand the node **Chapter - Generated Code**. By default, the report includes two sections, each containing one of two report components.
2. Expand the node **Section 1 — Code Generation Summary**.
3. Select **Code Generation Summary**. Options for the component are displayed in the properties pane.
4. Click **Help** to review the report customizations that you can make with the Code Generation Summary component. For this example, do not customize the component.
5. In the Report Explorer window, expand the node **Section 1 — Generated Code Listing**.
6. Select **Import Generated Code**. Options for the component are displayed in the properties pane.
7. Click **Help** to review the report customizations that you can make with the Import Generated Code component.

Generate the Report

After you adjust the report options, from the **Report Explorer** window, generate the report by clicking **File** > **Report**. A **Message List** dialog box opens, which displays messages that you can monitor as the report is generated. Model snapshots also appear during report generation. The **Message List** dialog box might be hidden behind other dialog boxes.



When the report is complete, open the report, `MyModelCGReport.rtf` in the folder `report_ex` (in this example).

Code Appearance in Embedded Coder

Add Custom Comments to Generated Code

You can include auto-generated comments in the generated code as described in "Configure Code Comments" (Simulink Coder). For ERT targets, include additional custom comments by setting parameters on the **Code Generation > Comments** pane in the Configuration Parameters dialog box. With these parameters, you can enable or suppress generation of descriptive information in comments for blocks and other model elements.

Goal	Specify
Include the text specified in the **Description** field of a block's Block Properties dialog box as comments in the code generated for each block.	**Simulink block descriptions**
Add a comment that includes the block name at the start of the code for each block.	**Simulink block descriptions**
Include the text specified in the **Description** field of a Simulink data object (such as a signal, parameter, data type, or bus) in the Simulink Model Explorer as comments in the code generated for each object.	**Simulink data object descriptions**
Include comments just above signals and parameter identifiers in the generated code as specified in the MATLAB or TLC function.	**Custom comments (MPT objects only)**
Include the text specified in the **Description** field in 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 "Generate Code for Models with Requirements Links" (Simulink Verification and Validation)).	**Requirements in block comments**

When you select **Simulink block descriptions**:

- The code generator includes strings for model parameters, block names, signal names, and Stateflow object names in the generated code comments. If those strings are unrepresented in the character set encoding for the model, the code generator replaces

the strings with XML escape sequences. For example, the code generator replaces the Japanese full-width Katakana letter ア with the escape sequence `&#x30A2;`. For more information, see "Internationalization and Code Generation" (Simulink Coder).

- The code generation software automatically inserts comments into the generated code for custom blocks. Therefore, you do not need to include block comments in the associated TLC file for a custom block.

  Note: If you have existing TLC files with manually inserted comments for block descriptions, the code generation process emits these comments instead of the automatically generated comments. Consider removing existing block comments from your TLC files. Manually inserted comments might be poorly formatted in the generated code and code-to-model traceability might not work.

- For virtual blocks or blocks that have been removed due to block reduction, comments are not generated.

For more information, see "Model Configuration Parameters: Code Generation Comments" (Simulink Coder).

Add Custom Comments for Variables in the Generated Code

To control code generation options for signals, states, and parameters in a model, you can create data objects in a workspace or data dictionary. You can generate comments in the code that help you to document the purpose and properties of the data in each object. Associate handwritten comments with each object, or write a function that generates comments based on the properties of the object.

For more information about data objects, see "Data Objects" (Simulink).

In this section...

Embed Handwritten Comments for Signals or Parameters

To embed handwritten comments in the generated code near the definition of a signal, state, or parameter:

1. Create a data object to represent a signal, state, or parameter. You can use a data object from any package. For example, use a data object of the classes `Simulink.Signal` or `Simulink.Parameter`, which are defined in the package `Simulink`.

   ```
   myParam = Simulink.Parameter(15.23);
   ```

2. Set the storage class of the data object so that optimizations do not eliminate the signal or parameter from the generated code. For example, use the storage class `ExportedGlobal`.

   ```
   myParam.StorageClass = 'ExportedGlobal';
   ```

3. Set the `Description` property of the object. The description that you specify appears in the generated code as lines of comments.

   ```
   myParam.Description = 'This parameter represents wind speed.';
   ```

4. Set **Configuration Parameters** > **Code Generation** > **System target file** to an ERT-based target such as `ert.tlc`.

   To generate comments from data object descriptions, you must use an ERT-based target.

5 Select **Configuration Parameters** > **Code Generation** > **Comments** > **Simulink data object descriptions**.

6 Generate code from the model. In the code, the data object description appears near the definition of the corresponding variable.

```
/* Exported block parameters */
real_T myParam = 15.23;     /* Variable: myParam
                             * Referenced by: '<S1>/Gain'
                             * This parameter represents wind speed.
                             */
```

Generate Dynamic Comments Based on Data Properties

You can generate dynamic comments that include the properties of the data object such as data type, units, and dimensions. If you change the properties of the data object in Simulink, the code generator maintains the accuracy of the comments. For example, this comment displays some of the property values for a data object named `MAP`:

```
/*     Unit:       psi                                          */
/*     Owner:                                                   */
/*     DefinitionFile: specialDef                               */
real_T MAP = 0.0;
```

1 Create a data object from the package `mpt` and apply a custom storage class to the object. The default storage class for objects that you create from the package `mpt` is the custom storage class `Global (Custom)`.

```
MAP = mpt.Signal;
```

To generate dynamic comments, you must use a data object from the package `mpt`, and you must apply a custom storage class to the object.

2 Write a MATLAB or TLC function that generates the comment text. For an example MATLAB function, see the function *`matlabroot`*`/toolbox/rtw/rtwdemos/rtwdemo_comments_mptfun.m`.

The function must accept three input arguments that correspond to `objectName`, `modelName`, and `request`. If you write a TLC file, you can use the library function `LibGetSLDataObjectInfo` to get the property values of the data object.

3 Save the function as a MATLAB file or a TLC file, and place the file in a folder that is on your MATLAB path.

4. In the model, select **Configuration Parameters > Code Generation > Comments > Custom comments (MPT objects only)**.
5. Set **Custom comments function** to the name of the MATLAB file or TLC file that you created.
6. Generate code from the model. The comments that your function generates appear near the code that represents each data object.

Limitations

- To generate comments by using the **Custom comments (MPT objects only)** and **Custom comments function** options, you must create data objects from the package `mpt`. The data objects must use a custom storage class.
- Only the custom storage classes from the `mpt` package that create unstructured variables support a custom comments function.

Related Examples

- "Add Custom Comments to Generated Code" on page 36-3
- "Control Data Representation by Applying Custom Storage Classes" on page 23-58

More About

- "Data Objects" (Simulink)
- "Introduction to Custom Storage Classes" on page 23-2
- "MPT Data Object Properties" on page 22-2

Add Global Comments

In this section...

The following examples show how to add a global comment to a Simulink model so that the comment text appears in the generated file or files where you want. Specify a template symbol name with a Simulink DocBlock, a Simulink annotation, or a Stateflow note. You can also use a sorted-notes capability that works with Simulink annotations or Stateflow notes (but not DocBlocks). For more information about template symbols, see "Template Symbols and Rules" on page 36-90.

Note Template symbol names `Description` and `ModifiedHistory` also are fields in the Model Properties dialog box. If you use one of these symbol names for global comment text, and its Model Properties field also has text in it, both names appear in the generated files.

Use a Simulink DocBlock to Add a Comment

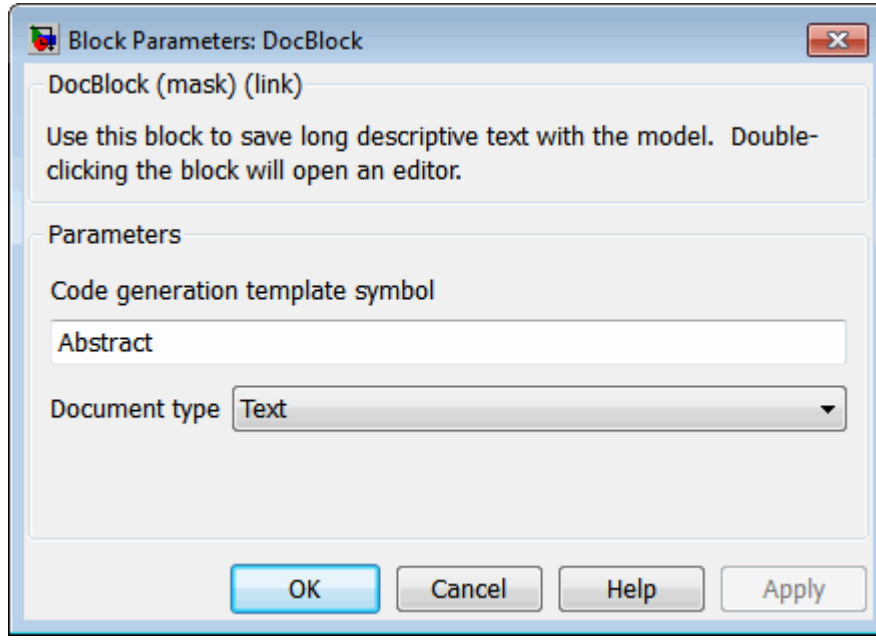
1. With the model open, from the **View** menu, select **Library Browser**.
2. Drag the DocBlock from **Model-Wide Utilities** in the Simulink library into the model.
3. Double-click the DocBlock and type the comment that you want in the editor. Save and close the editor.
4. Right-click the DocBlock and select **Mask > Mask Parameters**.
5. In the **Code generation template symbol** box, type one of the following:

   - `Abstract`
   - `Description`
   - `History`
   - `ModifiedHistory`

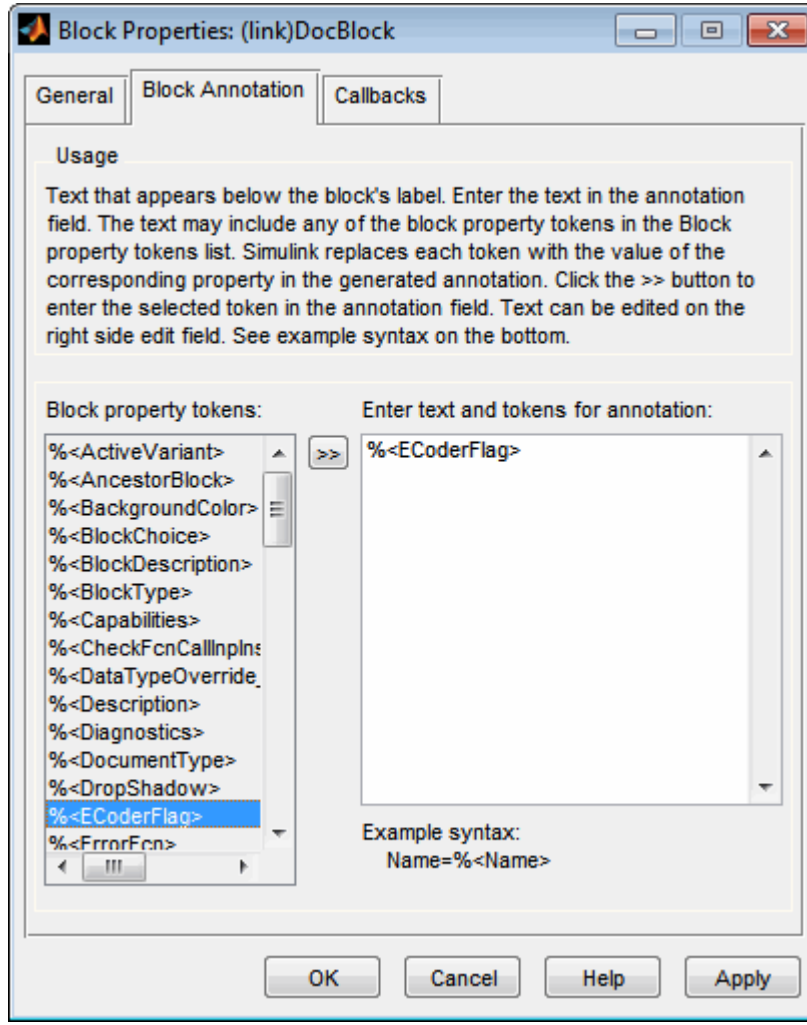
- `Notes`

Click **OK**. Template symbol names are case sensitive.

If you are using a DocBlock to add comments to your code, set the **Document type** to `Text`. If you set **Document type** to `RTF` or `HTML`, your comments will not appear in the code.



6 In the Block Properties dialog box, on the **Block Annotation** tab, select `%<ECoderFlag>` and click **OK**. The symbol name that you typed in the previous step now appears under the DocBlock in the model.



7 Save the model. After you generate code, the code generator places the comment in each generated file whose template has the symbol name that you typed. The code generator places the comment in the generated file at the location that corresponds to where the symbol name is located in the template file.

8 To add more comments to the generated files, repeat steps 1–7.

Use a Simulink Annotation to Add a Comment

1. Double-click the unoccupied area on the model where you want to place the comment. See "Describe Models Using Annotations" (Simulink).
2. Type `<S:Symbol_name>` followed by the comment. `Symbol_name` is one of the following:

   - `Abstract`
   - `Description`
   - `History`
   - `ModifiedHistory`
   - `Notes`

   For example, type `<S:Description>This is the description I want`. Template symbol names are case sensitive. (The `"S"` before the colon indicates "symbol.") If you want the code generator to sort multiple comments for the `Notes` symbol name, replace the next step with "Use Sorted Notes to Add Comments" on page 36-12.
3. Click outside the rectangle and save the model. After you generate code, the code generator places the comment in each generated file whose template has the symbol name that you typed. The code generator places the comment in the generated file at the location that corresponds to where the symbol name is located in the template file. If you want the code generator to sort multiple comments for the `Notes` symbol name, replace the next step with "Use Sorted Notes to Add Comments" on page 36-12.
4. To add one or more other comments to the generated files, repeat steps 1–3.

Use a Stateflow Note to Add a Comment

1. Right-click the unoccupied area on the Stateflow chart where you want to place the comment.
2. Select the annotation icon from the palette.
3. Type `<S:Symbol_name>` followed by the comment. `Symbol_name` is one of the following:

   - `Abstract`
   - `Description`

- `History`
- `ModifiedHistory`
- `Notes`

For example, type `<S:Description>This is the description I want`. Template symbol names are case sensitive. If you want the code generator to sort multiple comments for the `Notes` symbol name, replace the next step with "Use Sorted Notes to Add Comments" on page 36-12.

4 Click outside the note and save the model. After you generate code, the code generator places the comment in each generated file whose template has the symbol name that you typed. The code generator places the comment in the generated file at the location that corresponds to where the symbol name is located in the template file.

5 To add one or more other comments to the generated files, repeat steps 1–4.

Use Sorted Notes to Add Comments

The sorted-notes capability allows you to add automatically sorted comments to the generated files. The code generator places these comments in each generated file at the location that corresponds to where the `Notes` symbol is located in the template file.

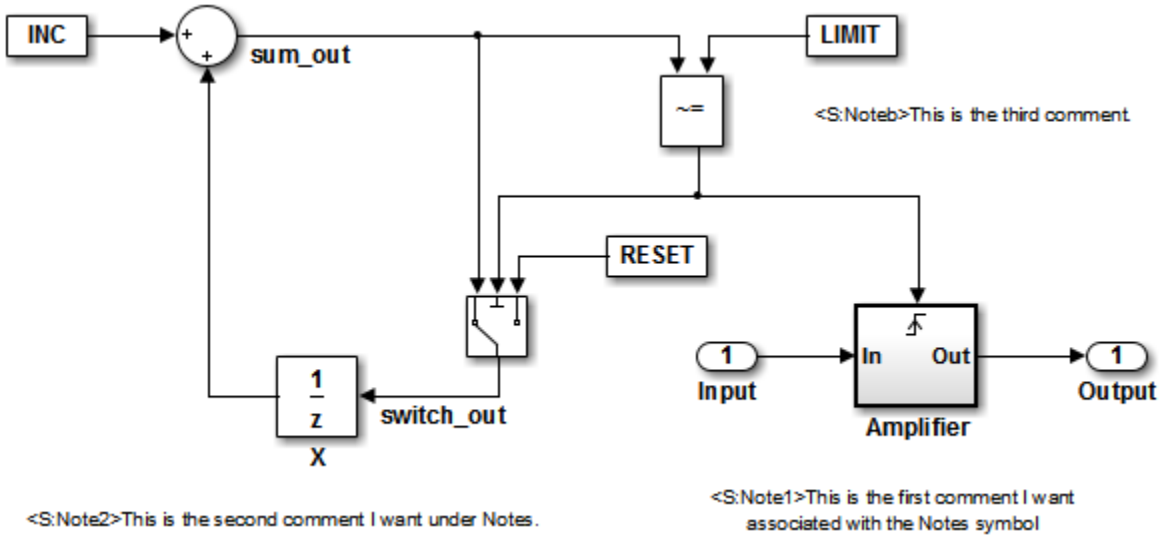
The code generator uses the following sorting order:

- Numbers before letters.
- Among numbers, 0 is first.
- Among letters, uppercase are before lowercase.

You can use sorted notes with a Simulink annotation or a Stateflow note, but not with a DocBlock.

- In the Simulink annotation or the Stateflow note, type `<S:NoteY>` followed by the first comment. `Y` is a number or a letter.
- Repeat for as many additional comments you want. Replace `Y` with a subsequent number or letter.

The figure illustrates sorted notes on a model, and where the code generator places each note in a generated file.



The relevant fragment from the generated file for this model is:

```
** NOTES

** Note1: This is the first comment I want
associated with the Notes symbol.
Note2: This is the second comment I want under Notes.
Noteb: This is the third comment.

**
```

Specify Comment Style

For ERT-based models, the comment style used in generated code is determined by the programming language selected for the model:

- C code uses `/*...*/` notation for both single-line and multiple-line comments.
- C++ code uses `//...` notation and contains only single-line comments.

If you have an Embedded Coder license, you can modify the comment style for generated code using the command-line parameter `CommentStyle`. The parameter takes the following values:

Value	Description
`Auto` (default)	For C, generate single or multiple-line comments delimited by `/*` and `*/`. For C++, generate single-line comments preceded by `//`.
`Multi-line`	Generate single or multiple-line comments delimited by `/*` and `*/`.
`Single-line`	Generate single-line comments preceded by `//`.

For example, the following command sets the comment style to single-line comments:

```
>> set_param('rtwdemo_counter','CommentStyle','Single-line')
```

Here is an example of code generated using the single-line comment style:

```
// Sum: '<Root>/Sum' incorporates:
//   Constant: '<Root>/INC'
//   UnitDelay: '<Root>/X'

rtb_sum_out = (uint8_T)(1U + rtwdemo_counter_DW.X);
```

Note: For C code generation, select `Single-line` only if your compiler supports it

Customize Generated Identifier Naming Rules

In this section...

For GRT and RSim targets, the code generator constructs identifiers for variables and functions in the generated code. For ERT targets, you can customize the naming of identifiers in the generated code by specifying parameters on the **Code Generation > Symbols** pane in the Configuration Parameters dialog box. You can also specify parameters that control identifiers generated from Simulink data objects. For detailed information about these parameters, see “Model Configuration Parameters: Code Generation Symbols” (Simulink Coder).

Apply Naming Rules to Identifiers Globally

Goal	Specify
Set the maximum number of characters that the code generator uses for function, `typedef`, and variable names (default 31) .	An integer value for the “Maximum identifier length” (Simulink Coder) parameter. For more information, see “Specify Identifier Length to Avoid Naming Collisions” (Simulink Coder). If you expect your model to generate lengthy identifiers (due to use of long signal or parameter names, for example), or if identifiers are mangled more than you expect, increase the value of this parameter.
Define a macro that specifies certain text included within generated identifiers for: • Global variables • Global types • Field names of global types • Subsystem methods • Subsystem method arguments • Local temporary variables • Local block output variables	A macro for the **Identifier format control** parameters. For more information, see “Identifier Format Control” on page 36-22. See also “Exceptions to Identifier Formatting Conventions” on page 36-33 and “Identifier Format Control Parameters Limitations” on page 36-34.

Goal	Specify
• Constant macros • Shared utilities	
Set the minimum number of characters that the code generator uses for the mangling text.	An integer value for the "Minimum mangle length" (Simulink Coder) parameter. For more information, see "Control Name Mangling in Generated Identifiers" on page 36-28
Control whether the software uses shortened names for system-generated identifiers.	`Shortened` for the "System-generated identifiers" (Simulink Coder) parameter. This setting: • Provides more space for user names. • Provides a more predictable and consistent naming system that uses camel case. • Does not include underscores or plurals. • Provides consistent abbreviations for both a type and a variable.
Control whether the generated code expresses scalar inlined parameter values as literal values or as macros.	The value `Literals` or `Macros` for the "Generate scalar inlined parameters as" (Simulink Coder) parameter. • `Literals`: If you set **Default parameter behavior** to `Inlined`, parameters are expressed as numeric constants. • `Macros`: Parameters are expressed as variables (with `#define` macros). This setting makes code more readable.

Apply Naming Rules to Simulink Data Objects

When your model uses Simulink data objects from the `Simulink` package, identifiers in generated code copy the names of the objects by default. For example, a `Simulink.Signal` object named `Speed` appears as the identifier `Speed` in generated code.

You can control these identifiers by specifying naming rules that are specific to Simulink data objects. On the **Code Generation > Symbols** pane of the Configuration

Parameters dialog box, adjust the settings in the **Simulink data object naming rules** section .

When you specify naming rules for generated code, follow ANSI C[5]/C++ rules for naming identifiers.

Specify Naming Rule Using a Function

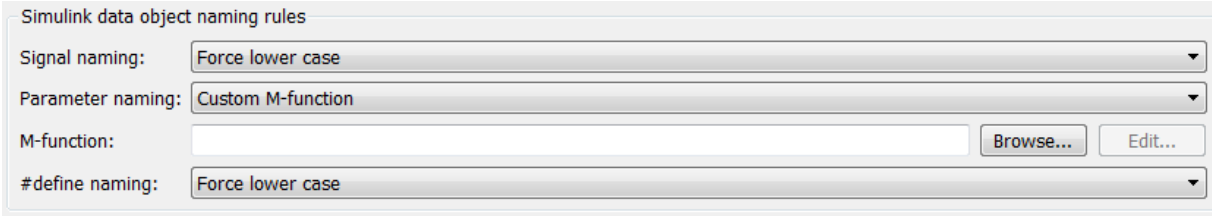
This example shows how to customize identifiers in generated code by defining a MATLAB function.

1. Write a MATLAB function that returns an identifier by modifying a data object name, and save the function in your working folder. For example, the following function returns an identifier name by appending the text `_param` to a data object name.

```
function revisedName = append_text(name, object)
% APPEND_TEXT: Returns an identifier for generated
% code by appending text to a data object name.
%
% Input arguments:
% name: data object name as spelled in model
% object: target data object
%
% Output arguments:
% revisedName: altered identifier returned for use in
% generated code.
%
%
text = '_param';

revisedName = [name,text];
```

2. Open the model `rtwdemo_namerules`.
3. Double-click the yellow box labeled **View Symbols Configuration** to open the **Code Generation** > **Symbols** pane in the Configuration Parameters dialog box.
4. From the **Parameter naming** (Simulink Coder) drop-down list, select **Custom M-function**.

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



5 In the **M-function** field, type the name of the file that defines the MATLAB function, `append_text.m`.

6 Click **Apply**.

7 Generate code for the model.

8 Inspect the code generation report to confirm the parameter object naming rule. For example, the generated file `rtwdemo_namerules.h` represents the parameter objects `G1`, `G2`, and `G3` with the variables `G1_param`, `G2_param`, and `G3_param`.

Specify Naming Rule for Storage Class `Define`

You can specify a naming rule that applies only to Simulink data objects whose storage class you set to `Define`. For these data objects, the specified naming rule overrides the other parameter and signal object naming rules. On the **Code Generation > Symbols** pane in the Configuration Parameters dialog box, adjust the **#define naming** (Simulink Coder) setting.

Override Data Object Naming Rules

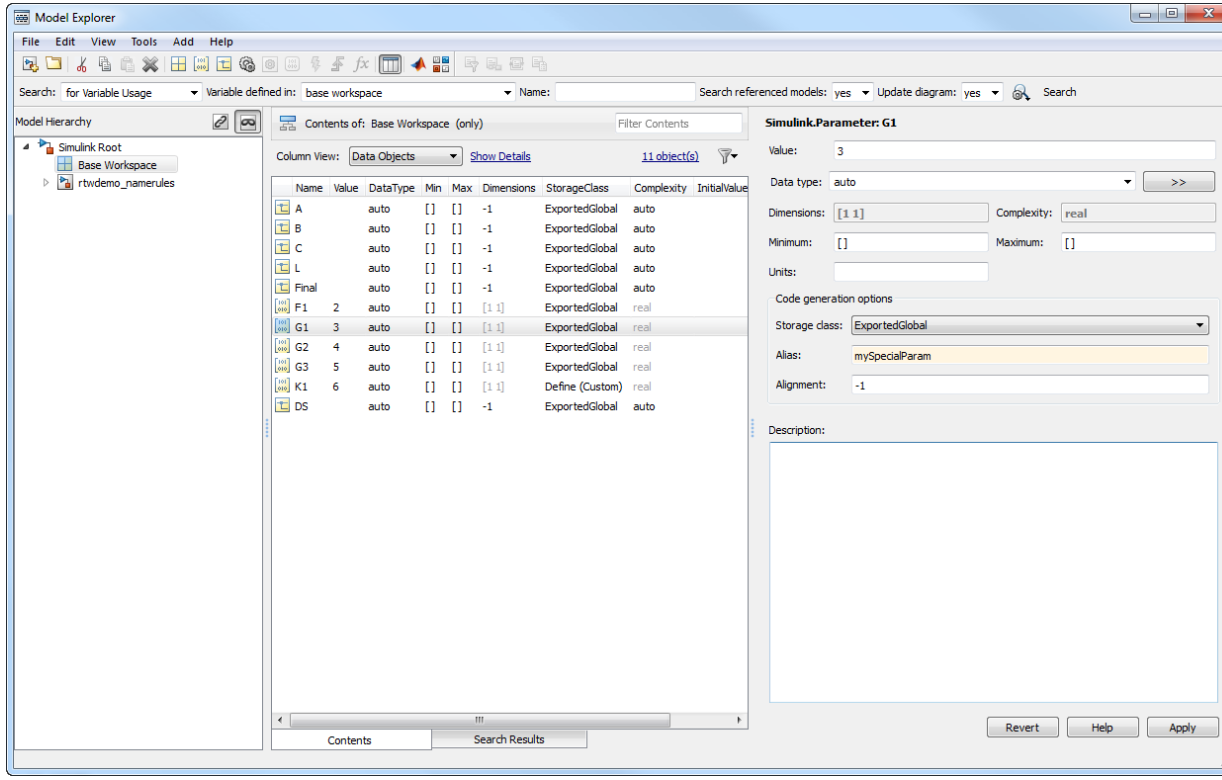
This example shows how to override a data object naming rule for a single data object.

You can override data object naming rules by specifying the `Alias` property of an individual Simulink data object. Generated code uses the text that you specify as the identifier to represent the data object, regardless of naming rules.

1 Open the model `rtwdemo_namerules`.

2 Open Model Explorer and navigate to the base workspace.

3 Click the parameter object `G1` and specify the `Alias` property as `mySpecialParam`. Click **Apply**.



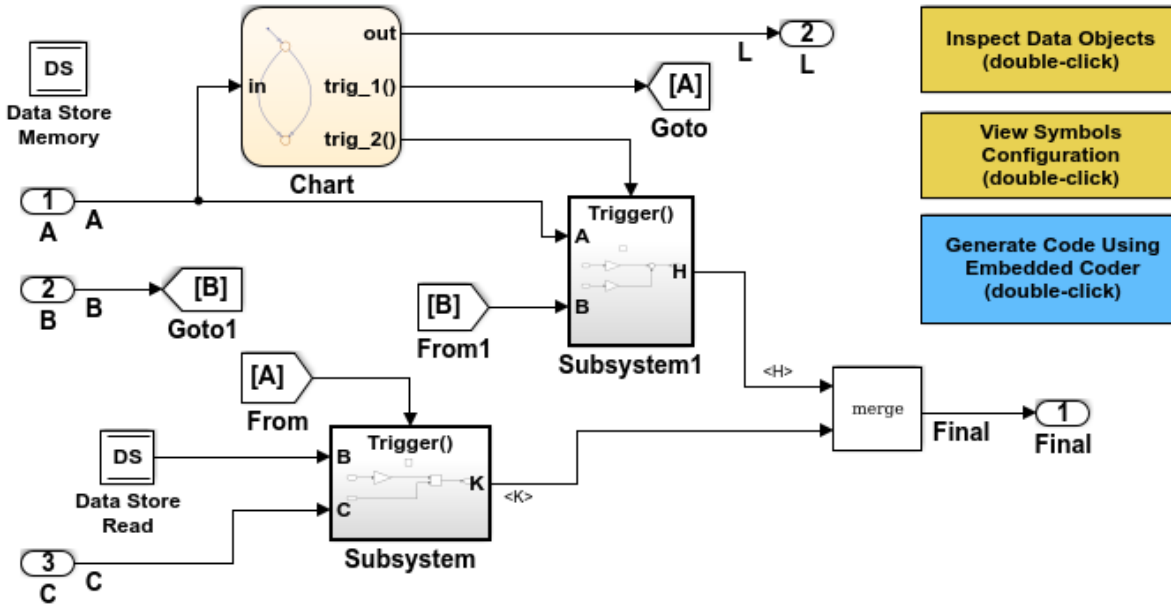
4 Generate code for the model.

5 In the code generation report, confirm the alias for the parameter object `G1`. The generated file `rtwdemo_namerules.h` represents `G1` with the variable `mySpecialParam`.

Apply Custom Naming Conventions to Identifiers

This example shows how to apply uniform naming rules for Simulink® data objects, including signals, parameters, and data store memory variables.

```
model='rtwdemo_namerules';
open_system(model)
```



Description
Many organizations employ coding standards that include naming rules for variables. This model shows how to apply uniform naming rules for Simulink data objects, including signals, parameters, and data store memory variables. Predefined conventions exist to apply upper and lower case naming, and arbitrary custom rules can be defined in a user-supplied MATLAB script.

Instructions
1. Double-click the yellow View Symbols Configuration button to inspect the "Simulink data object naming rules."
2. Double-click the yellow Inspect Data Objects button to inspect the data objects for this model.
3. Generate code using the blue button in the upper right of the diagram. An HTML report automatically appears.
4. Inspect the data that is defined toward the top of rtwdemo_namerules.c.

Note
The model is configured to "Force lower case." Therefore, the variables in the code are defined in lower case even though the data is declared upper case in the model.


```
% Cleanup
rtwdemoclean;
```

```
close_system(model,0)
```

See Also

"Signal naming" (Simulink Coder)

Identifier Format Control

You can customize generated identifiers by specifying the **Identifier format control** parameters on the **Code Generation > Symbols** pane in the Configuration Parameters dialog box. For each parameter, you can enter a macro that specifies whether, and in what order, certain text is included within generated identifiers. For example, you can specify that the root model name be inserted into each identifier using the `$R` token.

The macro can include:

- Valid tokens, which are listed in Identifier Format Tokens. You can use or omit tokens depending on what you want to include in the identifier name. The **Shared utilities** parameter requires you to specify the checksum token, `$C` . The other parameters require the mangling token, `$M`. For more information, see "Control Name Mangling in Generated Identifiers" on page 36-28. The mangling token is subject to the use and ordering restrictions noted in Identifier Format Control Parameter Values.
- Token decorators, which are listed in "Control Case with Token Decorators" on page 36-25. You can use token decorators to control the case of generated identifiers for each token.
- Valid C or C++ language identifier characters (`a-z, A-Z, _ , 0-9`).

The build process generates each identifier by expanding tokens and inserting the resultant text into the identifier. The tokens are expanded in the order listed in Identifier Format Tokens. Groups of characters are inserted in the positions that you specify around tokens directly into the identifier. Contiguous token expansions are separated by the underscore (_) character.

Identifier Format Tokens

Token	Description
`$C`	This token is required for **Shared utilities**. If the identifier exceeds the **Maximum identifier length**, the code generator inserts an 8-character checksum to avoid naming collisions. The position of the `$C` token in the **Identifier format control** parameter specification determines the position of the checksum in the generated identifier. For example, if you use the specification `$N$C`, the checksum is appended to the end of the identifier. This token is available only for shared utilities.
`$M`	This token is required. If necessary, the code generator inserts name-mangling text to avoid naming collisions. The position of the `$M` token

Token	Description
	in the **Identifier format control** parameter specification determines the position of the name-mangling text in the generated identifier. For example, if you use the specification `$R$N$M`, the name-mangling text is appended (if required) to the end of the identifier. For more information, see "Control Name Mangling in Generated Identifiers" on page 36-28 .
`$U`	Insert text that you specify for the `$U` token. Use the **Custom token text** parameter to specify this text. This parameter is on the **All Parameters** tab of the Configuration Parameters dialog box. See "Custom token text".
`$F`	Insert method name (for example, `_Update` for update method). This token is available only for subsystem methods.
`$N`	Insert name of object (block, signal or signal object, state, parameter, shared utility function or parameter object) for which identifier is being generated.
`$R`	Insert root model name into identifier, replacing unsupported characters with the underscore (_) character. When you use referenced models, this token is required in addition to `$M` (see "Avoid Identifier Name Collisions with Referenced Models" on page 36-30). **Note:** This token replaces the **Prefix model name to global identifiers** option in previous releases.
`$H`	Insert tag indicating system hierarchy level. For root-level blocks, the tag is the text `root_`. For blocks at the subsystem level, the tag is of the form `sN_`. `N` is a unique system number assigned by the Simulink software. This token is available only for subsystem methods and field names of global types. **Note:** This token replaces the **Include System Hierarchy Number in Identifiers** option in previous releases.
`$A`	Insert data type acronym (for example, `i32` for integers) to signal and work vector identifiers. This token is available for local block output variables, local temporary variables, and field names of global types. **Note:** This token replaces the **Include data type acronym in identifier** option in previous releases.
`$I`	• Insert `u` if the argument is an input.

Token	Description
	• Insert `y` if the argument is an output. • Insert `uy` if the argument is an input and output. For example, `rtu_` for an input argument, `rty_` for an output argument, and `rtuy_` for an input and output argument. This token is available only for subsystem method arguments.

Identifier Format Control Parameter Values lists the default macro value, 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 (Simulink Coder)	`rt$N$M`	`$R`, `$N`, `$M`,`$U`	`$F`, `$H`, `$A`, and `$I` are not allowed.
Global types (Simulink Coder)	`$N$R$M_T`	`$N`, `$R`, `$M`, `$U`	`$F`, `$H`, `$A`, and `$I` are not allowed.
Field name of global types (Simulink Coder)	`$N$M`	`$N`, `$M`, `$H`, `$A`, `$U`	`$R`, `$F`, and `$I` are not allowed.
Subsystem methods (Simulink Coder)	`$F$N$M`	`$R`, `$N`, `$M`, `$F`, `$H`, `$U`	`$F` and `$H` are empty for Stateflow functions; `$A` and `$I` are not allowed.
Subsystem method arguments (Simulink Coder)	`rt$I$N$M`	`$N`, `$M`, `$I`, `$U`	`$R`, `$F`, `$H`, and `$A` are not allowed.
Local temporary variables (Simulink Coder)	`$N$M`	`$N`, `$M`, `$R`, `$A`, `$U`	`$F`, `$H`, and `$I` are not allowed.
Local block output variables (Simulink Coder)	`rtb_$N$M`	`$N`, `$M`, `$A`, `$U`	`$R`, `$F`, `$H`, and `$I` are not allowed.
Constant macros (Simulink Coder)	`$R$N$M`	`$R`, `$N`, `$M`, `$U`	`$F`, `$H`, `$A`, and `$I` are not allowed.

Parameter	Default Value	Supported Tokens	Restrictions
Shared utilities	`$N$C`	`$N`, `$C`, `$N`, `$U`	`$C` is required. `$M`, `$F`, `$H`, `$A` , and `$I` are not allowed.
EMX array utility functions identifier format (Simulink Coder)	`emx$M$N`	`$M`, `$N`,`$R`	`$C`, `$U`, `$F`, `$H`, `$A` , and `$I` are not allowed.
EMX array types identifier format (Simulink Coder)	`emxArray_ $M$N`	`$M`, `$N`,`$R`	`$C`, `$U`, `$F`, `$H`, `$A` , and `$I` are not allowed.

Non-ERT-based targets (such as the GRT target) implicitly use a default `$R$N$M` specification. This default specification consists of the root model name, followed by the name of the generating object (signal, parameter, state, and so on), followed by name-mangling text.

For limitations that apply to **Identifier format control** parameters, see "Exceptions to Identifier Formatting Conventions" on page 36-33 and "Identifier Format Control Parameters Limitations" on page 36-34.

Control Case with Token Decorators

On the **Code Generation > Symbols** pane, you can use token decorators to control the case of generated identifiers. Place a decorator immediately after the target token and enclose the decorator in square brackets `[ ]`. For example, you can set **Global variables** to `$R[uL]$N$M`, which capitalizes the first letter of the model name and forces the remaining characters in the model name to lowercase.

The table shows how to manipulate the expansion of the `$R` token for a model whose name is `modelName`.

Desired Expansion	Description	Token and Decorator
`ModelName`	First letter of model name is uppercase. Remaining characters are not modified.	`$R[u]`

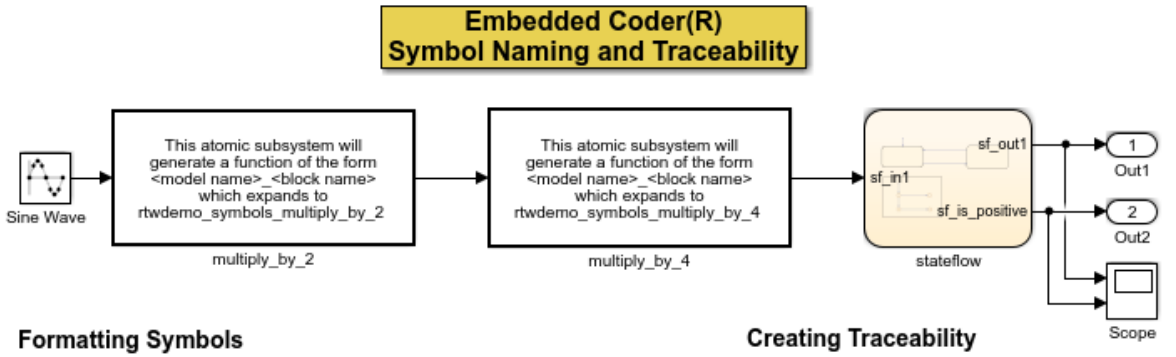
Desired Expansion	Description	Token and Decorator
`Modelname`	First letter of model name is uppercase. Remaining characters are lowercase.	`$R[uL]`
`MODELNAME`	All characters are uppercase.	`$R[U]`
`modelname`	All characters are lowercase.	`$R[L]`
`mODELNAME`	First letter of model name is lowercase. Remaining characters are uppercase.	`$R[lU]`
`modelName`	First letter of model name is lowercase. Remaining characters are not modified.	`$R[l]`

When you use a decorator, the code generator removes the underscore character (_) that appears between tokens by default. However, you can append each decorator with an underscore: `$R[U_]$N`. For example, if you set the **Global variables** parameter to `$R[u_]$N[uL]$M` for a model named `modelName` and a `DWork` structure represented by `DW`, the result is `ModelName_Dw`.

Control Formatting of Identifiers

This example shows how you can customize generated identifiers by specifying the **Identifier format control** parameters on the **Code Generation > Symbols** pane in the Configuration Parameters dialog box.

```
model='rtwdemo_symbols';
open_system(model)
```



Formatting Symbols

Symbols in Embedded Coder(R) can be formatted via the "Identifier format control" option group. These options can be composed of predefined macros and C-language literal characters.

Common macros include
- **$R** = Root model name
- **$N** = Name of object (block, signal, state, etc.)
- **$M** = Mangle used to uniquify symbol

For a complete list of macros and the rules by which they are expanded, see the Symbol format help link below.

Different identifier format control strings control different types of identifiers. Double-click the yellow buttons at the bottom to modify the format control strings for global variable names and global type names. Double-click the blue button to generate and inspect code for the change.

Double-click to view symbol format

Double-click to view symbol format help

Global variable names contain model name. (double-click to toggle)

Global structure type names contain model name. (double-click to toggle)

Creating Traceability

One aspect of traceability is making sure that incremental revisions to a model have minimal impact on the symbol names that appear in generated code. There are two ways of achieving this in Simulink:

1) Name objects in Simulink (blocks, signals, states, etc.) as uniquely as possible.
2) Make use of name mangling when conflicts cannot be avoided.

For a complete discussion of how name mangling can be used to create traceable code, see the help link below.

Double-click to view mangle settings

Double-click to view traceability help

Generate Code Using Embedded Coder (double-click)


```
% Cleanup
rtwdemoclean;
close_system(model,0)
```

Control Name Mangling in Generated Identifiers

The position of the `$M` token in the **Identifier format control** parameter specification determines the position of the name-mangling text in the generated identifiers. For example, if you use the specification `$R$N$M`, the name-mangling text is appended (if required) to the end of the identifier. For more information, see "Identifier Format Control" on page 36-22.

Name-Mangling Text Per Object

Object Type	Source of Mangling Text
Block diagram	Name of block diagram
Simulink block	Simulink identifier (for details, see "Locate Diagram Components Using Simulink Identifiers" (Simulink))
Simulink parameter	Full name of parameter owner (model or block) and parameter name
Simulink signal	Signal name, full name of source block, and port number
Stateflow objects	Complete path to Stateflow block and Stateflow computed name (unique within chart)

The length of the name-mangling text is specified by the **Minimum mangle length** (Simulink Coder) 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 value. A value of `4` allows for over 1.5 million collisions for a particular identifier before the mangle length is increased.

Minimize Name Mangling

The length of generated identifiers is limited by the **Maximum identifier length** (Simulink Coder) parameter. When a name collision exists, the `$M` token is expanded to the minimum number of characters required to avoid the collision. Other tokens are expanded in the order listed in Identifier Format Tokens. If the **Maximum identifier length** is not large enough to accommodate full expansions of the other tokens, partial expansions are used. To avoid partial expansions, it is good practice to:

- Avoid name collisions. One way to avoid name collisions is to not use 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 that you expect to generate.
- Set the **Maximum identifier length** parameter to reserve at least three characters for the name-mangling text. The length of the name-mangling text increases as the number of name collisions increases.

  If changes to the model create more or fewer collisions, existing name-mangling text increases or decreases in length. If the length of the name-mangling text increases, additional characters are appended to the existing text. For example, the mangling text `'xyz'` can change to `'xyzQ'`. For fewer collisions, the name-mangling text `'xyz'` changes to `'xy'`.

Avoid Identifier Name Collisions with Referenced Models

Within a model that uses referenced models, collisions between the names of the models are not allowed. When generating code from a model that uses model referencing:

- You must include the `$R` token in the **Identifier format control** parameter specifications (in addition to the `$M` token).
- The **Maximum identifier length** must be large enough to accommodate full expansions of the `$R` and `$M` tokens. If **Maximum identifier length** is too small, a code generation error occurs.

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.

If your model contains two referenced models with the same input or output port names, and one of the referenced models contains an atomic subsystem with "Function packaging" (Simulink) set to `Nonreuseable function`, a name conflict can occur and the build process produces an error.

Use Model Advisor to Detect Identifier Names Changed During Code Generation

For a referenced model, if the following **Configuration Parameters > Code Generation > Symbols** parameters have settings that do not contain a `$R` token (which represents the name of the reference model), code generation prepends the `$R` token to the identifier format.

- **Global variables**
- **Global types**
- **Subsystem methods**
- **Constant macros**

You can use the Model Advisor to identify referenced models in a model referencing hierarchy for which code generation changes these configuration parameter settings.

1. In the Simulink Editor, select **Analysis > Model Advisor**.
2. Select **By Task**.

3 Run the **Check code generation identifier formats used for model reference** check.

Maintain Traceability for Generated Identifiers

To verify your model, you can trace back and forth between generated identifiers and corresponding entities within the model. To maintain traceability, it is important that incremental revisions to a model have minimal impact on the identifier names that appear in generated code. There are two ways to minimally impact the identifier names:

- Choose unique names for Simulink objects (blocks, signals, states, and so on) as much as possible.
- Use name mangling when conflicts cannot be avoided.

The position of the name-mangling text is specified by the placement of the `$M` token in the **Identifier format control** parameters. Mangle characters consist of alphanumeric characters that are unique to each object. For more information, see “Control Name Mangling in Generated Identifiers” on page 36-28.

Exceptions to Identifier Formatting Conventions

There are some exceptions to the identifier formatting conventions described in "Identifier Format Control" on page 36-22.

- Type name generation: name mangling conventions do not apply to type names (that is, `typedef` statements) generated for global data types. If the `$R` token is included in the **Identifier format control** parameter specification, the model name is included in the `typedef`. When generating type definitions, the **Maximum identifier length** parameter is not respected.
- Non-`Auto` storage classes: the **Identifier format control** parameters specification does not affect objects (such as signals and parameters) that have a storage class other than `Auto` (such as `ImportedExtern` or `ExportedGlobal`).
- For shared utilities, code generation inserts the checksum specified by `$C` to prevent name collisions in the following situations:
  - `$C` is specified without `$N`.
  - The length of `$N` plus the length of the text that you specify exceeds the **Maximum identifier length**. Code generation truncates `$N` and inserts an 8-character checksum where you specified `$C` in the formatting scheme.

.

Identifier Format Control Parameters Limitations

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

- The following autogenerated identifiers currently do not fully comply with the setting of the **Maximum identifier length** parameter on the **Code Generation > Symbols** pane of the Configuration Parameters dialog box.
  - Model methods
    - The applicable format scheme is `$R$F`, and the longest `$F` is `_derivatives`, which is 12 characters long. The model name can be up to 19 characters without exceeding the default **Maximum identifier length** of 31.
  - Local functions generated by S-functions or by add-on products such as DSP System Toolbox that rely on S-functions
  - Local variables generated by S-functions or by add-on products such as DSP System Toolbox that rely on S-functions
  - `DW` identifiers generated by S-functions in referenced models
  - Fixed-point shared utility macros or shared utility functions
  - Simulink `rtm` macros
    - Most are within the default **Maximum identifier length** of 31, but some exceed the limit. Examples are `RTMSpecAccsGetStopRequestedValStoredAsPtr`, `RTMSpecAccsGetErrorStatusPointer`, and `RTMSpecAccsGetErrorStatusPointerPointer`.
  - Define protection guard macros
    - Header file guards, such as `_RTW_HEADER_$(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
  - Reentrant model function arguments

  - Local functions generated by S-functions or by add-on products such as DSP System Toolbox that rely on S-functions
  - Local variables generated by S-functions or by add-on products such as DSP System Toolbox that rely on S-functions
  - Fixed-point shared utility macros or shared utility functions
  - Include header guard macros
- The following external identifiers that are unknown to the Simulink software might conflict with autogenerated identifiers.
  - Identifiers defined in custom code
  - Identifiers defined in custom header files
  - Identifiers introduced through a non-ANSI C standard library
  - Identifiers defined by custom TLC code
- Identifiers generated for simulation targets might 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.
- Identifiers generated using a model name and bus object data type name, which are both long names, might exceed the **Maximum identifier length**. For example, a ground value variable name is generated as <*model_name*>`_rtZ`<*bus_name*>. If the *model_name* and *bus_name* are close to the maximum identifier length, the name exceeds the maximum identifier length.

Control Code Style

In this section...

You can change the code style, cast expressions, and indentation of your generated code to conform to certain coding standards. Modify style options by setting parameters on the **Code Generation > Code Style** pane.

In the generated code, you can control the following style aspects:

- Level of parenthesization, see “Control Parentheses in Generated Code” on page 36-37.
- Order of operands in expressions, see “Optimize Code by Reordering Commutable Operands” on page 36-39.
- Empty primary condition expressions in `if` statements, see “Preserve condition expression in if statement”.
- Whether to generate code for `if-elseif-else` decision logic as `switch-case` statements, see “Convert if-elseif-else patterns to switch-case statements”.
- Whether to include the `extern` keyword in function declarations, see “Preserve extern keyword in function declarations”.
- Whether to generate `default` cases for `switch-case` statements in the code for Stateflow charts, see “Suppress Generation of Default Cases for Unreachable Stateflow Switch Statements” on page 36-40.
- Whether to replace multiplications by powers of two with signed bitwise shifts, see “Replace Multiplication by Powers of Two with Signed Bitwise Shifts” on page 36-43. Whether to allow right shifts on signed integers, see “Generate Code with Right Shifts on Signed Integers” on page 36-45. Some coding standards, such

as MISRA, do not allow bitwise operations on signed integers. Clearing this option increases the likelihood of generating MISRA C:2012 compliant code.

- Cast expressions, see “Control Cast Expressions in Generated Code” on page 36-48.
- Indentation style, see “Control Indentation Style in Generated Code” on page 36-46.

Control Parentheses in Generated Code

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 only increase the readability of the code. Optional C parentheses vary between two stylistic extremes:

- Include the minimum parentheses required by C syntax and 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.

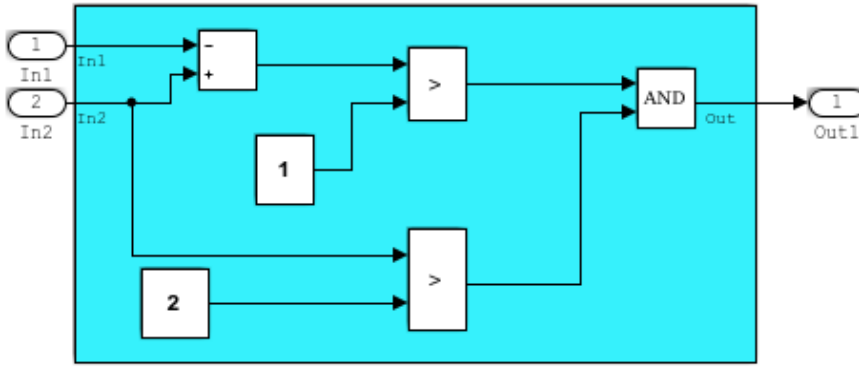
Understanding code with minimum parentheses can require applying nonobvious precedence rules. Maximum parentheses can hinder code reading by belaboring obvious precedence rules. Various parenthesization standards exist that specify one or the other extreme, or define an intermediate style useful to people who read code.

For more information on this parameter, see “Parentheses level”.

Control Use of Parentheses

This example shows that Embedded Coder® provides three levels of control for parentheses in the generated code.

```
model='rtwdemo_parentheses';
open_system(model)
```



View Code Style Configuration (double-click)

Generate Code Using Embedded Coder (double-click)

Parentheses level "Minimum": `Out = In2 - In1 > 1.0 && In2 > 2.0;`

Parentheses level "Nominal": `Out = ((In2 - In1 > 1.0) && (In2 > 2.0));`

Parentheses level "Maximum": `Out = (((In2 - In1) > 1.0) && (In2 > 2.0));`

Description
Embedded Coder(R) provides three levels of control for parentheses in the generated code. The control level "Minimum" generates the most compact code, with the smallest number of parentheses allowed under C syntax rules. The control level "Nominal" adds parentheses as needed to optimize readability. The control level "Maximum" generates the largest number of parentheses. The parentheses explicitly define the sequence of operations in an expression without relying on C precedence rules. Specifying control level "Maximum" provides full compliance with the MISRA-C(TM) standard.

Instructions
1. View and set the parentheses level by double-clicking the yellow button to the right.
2. Generate code by double-clicking the blue button to the right.
   An HTML report is displayed automatically.
3. Inspect the generated source files, comparing them with the three code examples above.


```
rtwdemoclean;
```

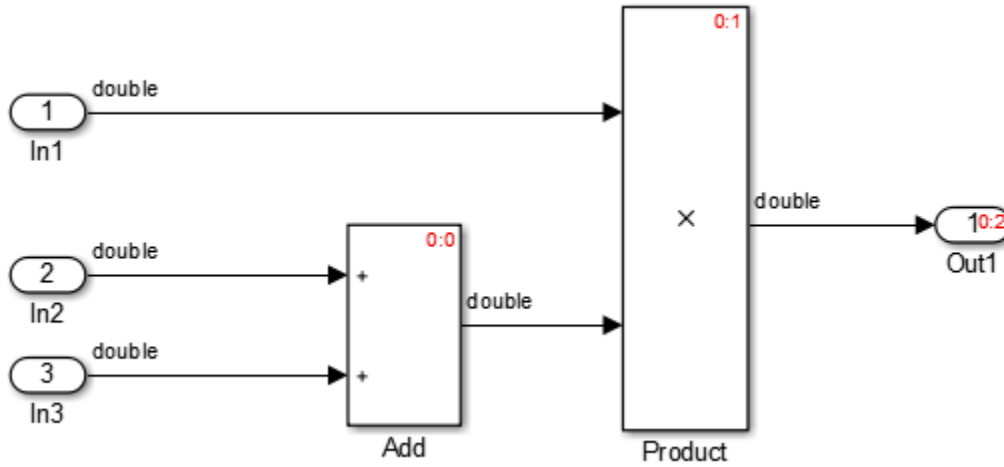
```
close_system(model,0)
```

Optimize Code by Reordering Commutable Operands

This example shows how to reorder commutable operands to make expressions left-recursive. This optimization improves code efficiency.

Example Model

To reorder commutable operands, create the following model and name it `operand_order`. In this model, the output signal is the result of multiplying the signal from Inport block `In1` by the sum of the signals from Inport blocks `In2` and `In3`.



Generate Code

1. Open the Model Configuration Parameters dialog box. On the **Code Style** tab, select the **Preserve operand order in expression** parameter.
2. Generate code for the model.

In the `operand_order.c` file, the `operand_order_step` function contains the following code:

```
operand_order_Y.Out1 = operand_order_U.In1 * (operand_order_U.In2 +
```

```
    operand_order_U.In3);
```

The code generator preserves the specified expression order in the model. Preserving the specified expression order increases the readability of the code for code traceability purposes.

Generate Code with Optimization

1. Open the Model Configuration Parameters dialog box. On the **Code Style** tab, clear the **Preserve operand order in expression** parameter.
2. Generate code for the model.

In the `operand_order.c` file, the `operand_order_step` function contains the following code:

```
operand_order_Y.Out1 = (operand_order_U.In2 + operand_order_U.In3) *
    operand_order_U.In1;
```

The code generator optimizes the code by reordering the commutable operands to make the expression left-recursive. Left-recursive expressions improve code efficiency.

For more information on the **Preserve operand order in expression** parameter, see "Preserve operand order in expression".

Suppress Generation of Default Cases for Unreachable Stateflow Switch Statements

This example shows how to specify whether to generate default cases for switch-case statements in the code for Stateflow charts. Generated code that does not contain default cases conserves ROM consumption and enables better code coverage because every branch in the generated code is falsifiable.

Some coding standards, such as MISRA, require the default case for switch-case statements. If you want to increase your chances of producing MISRA C compliant code, generate default cases for unreachable Stateflow switch statements.

Example

Figures 1, 2, and 3 show relevant portions of the `sldemo_fuelsys` model, a closed-loop system containing a plant and controller. The Air-fuel rate controller logic is a Stateflow chart that specifies the different operation modes.

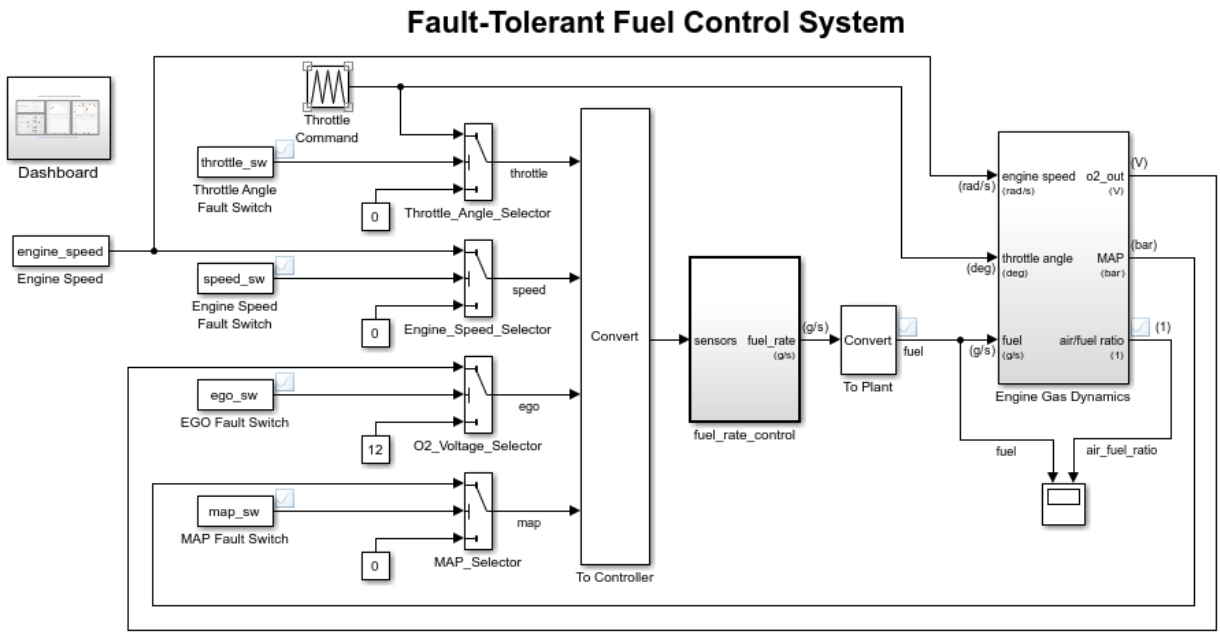


Figure 1: Top-level model of the plant and controller

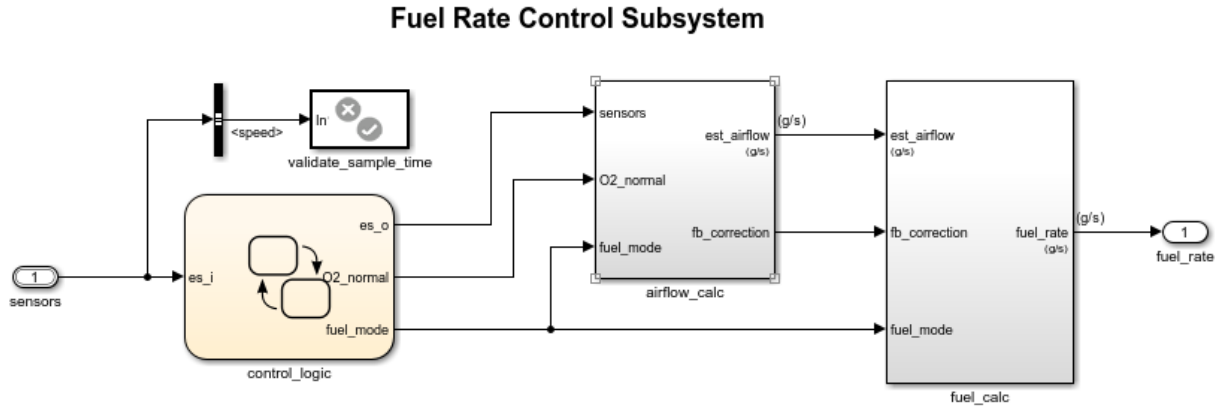


Figure 2: Fuel rate controller subsystem

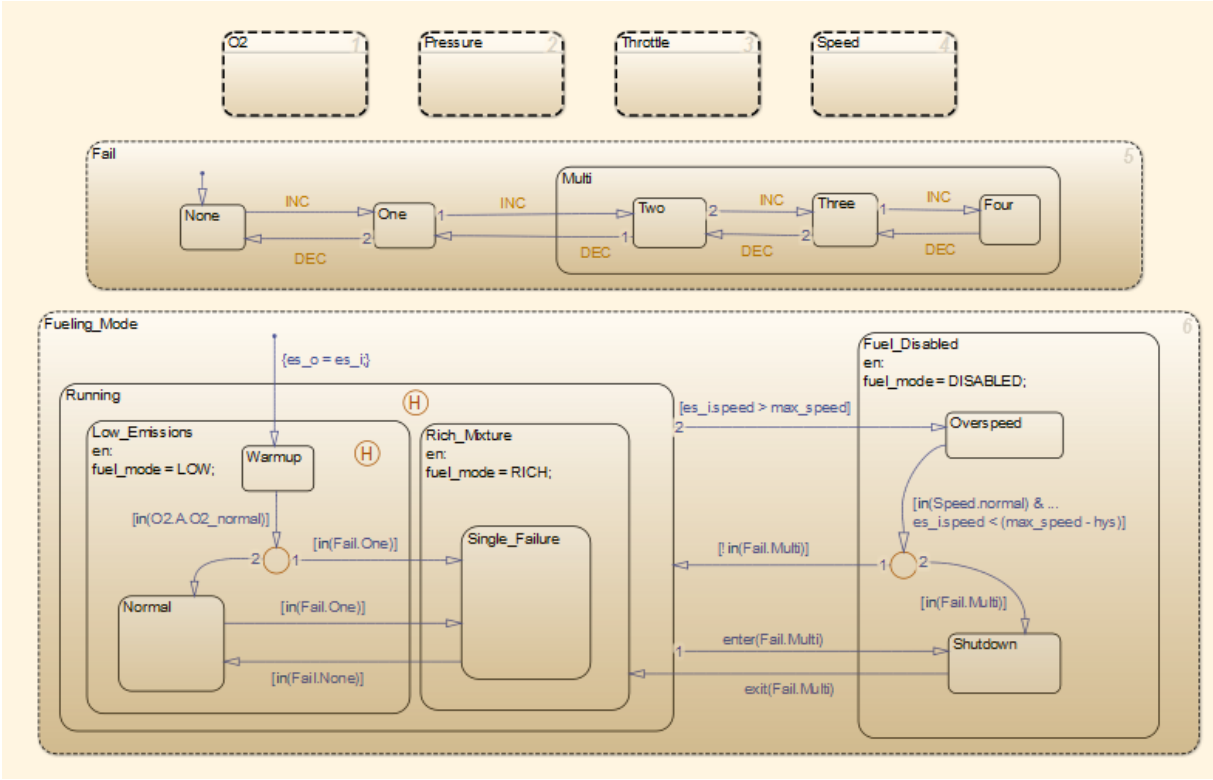


Figure 3: Fuel rate controller logic

Generate Code with Default Cases for Unreachable Stateflow Switch Statements

1. In the MATLAB Command Window, to open `sldemo_fuelsys` via `rtwdemo_fuelsys` enter:

   ```
   rtwdemo_fuelsys
   ```

2. Open the Model Configuration parameters dialog box. On the **Code Generation** > **Code Style** tab, clear the **Suppress generation of default cases for Stateflow statements if unreachable** parameter.

3. In the MATLAB Command Window, to build the model, enter:

   ```
   rtwbuild('sldemo_fuelsys/fuel_rate_control');
   ```

For the different operation modes, the `fuel_rate_control.c` file contains default cases for unreachable switch statements. For example, for the Shutdown operation mode, the generated code contains this default statement:

```
default:
 /* Unreachable state, for coverage only */
 rtDWork.bitsForTID0.is_Fuel_Disabled = IN_NO_ACTIVE_CHILD;
 break;
```

For the Warmup operation mode, the generated code contains this default statement:

```
default:
 /* Unreachable state, for coverage only */
 rtDWork.bitsForTID0.is_Low_Emissions = IN_NO_ACTIVE_CHILD;
 break;
```

Suppress Default Cases for Unreachable Stateflow Switch Statements

1. Open the Configuration Parameters dialog box. On the **Code Generation** > **Code Style** tab, select the **Suppress generation of default cases for Stateflow statements if unreachable** parameter.
2. Build the model.

Read through the `fuel_rate_control.c` file. The default cases for unreachable switch statements are not in the generated code.

For more information on the **Suppress generation of default cases for Stateflow statements if unreachable** parameter, see "Suppress generation of default cases for Stateflow switch statements if unreachable".

Replace Multiplication by Powers of Two with Signed Bitwise Shifts

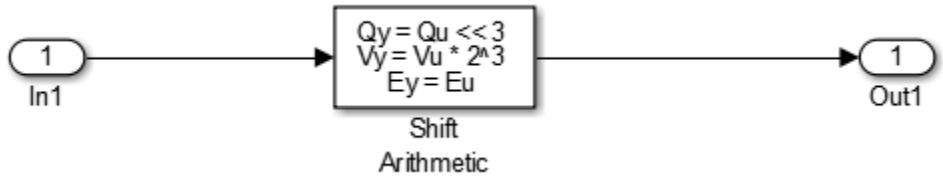
This example shows how to generate code that replaces multiplication by powers of two with signed bitwise shifts. Code that contains bitwise shifts is more efficient than code that contains multiplication by powers of two.

Some coding standards, such as MISRA, do not allow bitwise operations on signed integers. If you want to increase your chances of producing MISRA C compliant code, do not replace multiplication by powers of two with bitwise shifts.

Example

To replace multiplication by powers of two with bitwise shifts, create the following model. In this model, a signal of **Data type** `int16` feeds into a Shift Arithmetic block. In the

Shift Arithmetic Block Parameters dialog box, the **Bits to shift** > **Direction** parameter is set to `Left`. The **Bits to shift** > **Number** parameter is set to `3`. This parameter corresponds to a value of 8, or raising 2 to the power of 3.



Generate Code with Signed Bitwise Shifts

1. Open the Model Configuration Parameters dialog box and select the **Code Style** tab. The **Replace multiplications by powers of two with signed bitwise shifts** parameter is on by default.
2. Generate code for the model.

In the `bitwise_multiplication.c` file, the `bitwise_multiplication` step function contains this code:

```
bitwise_multiplication_Y.Out1 = (int16_T)(bitwise_multiplication_U.In1 << 3);
```

The signed integer, `bitwise_multiplication_U.In1`, is shifted three bits to the left.

Generate Code with Multiplication by Powers of Two

1. Open the Model Configuration Parameters dialog box and select the **Code Style** tab.
2. Clear the **Replace multiplications by powers of two with signed bitwise shifts** parameter.
3. Generate code for the model.

In the `bitwise_multiplication.c` file, the `bitwise_multiplication` step function contains this code:

```
bitwise_multiplication_Y.Out1 = (int16_T)(bitwise_multiplication_U.In1 * 8);
```

The signed integer `bitwise_multiplication_U.In1` is multiplied by `8`.

For more information on the **Replace multiplications by powers of two with signed bitwise shifts** parameter, see "Replace multiplications by powers of two with signed bitwise shifts".

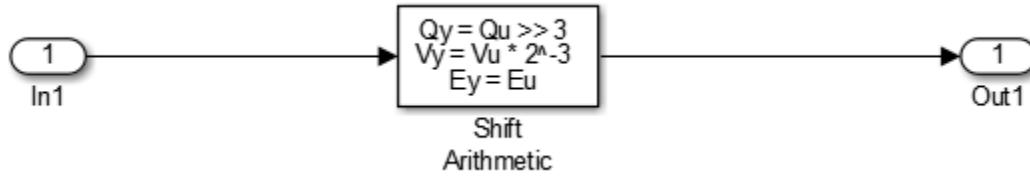
Generate Code with Right Shifts on Signed Integers

This example shows how to control whether the generated code contains right shifts on signed integers. Generated code that does not contain right shifts on signed integers first casts the signed integers to unsigned integers, and then right shifts the unsigned integers.

Some coding standards, such as MISRA, do not allow right shifts on signed integers because different hardware can store negative integers differently. For negative integers, you can get different answers depending on the hardware. If you want to increase your chances of producing MISRA C compliant code, do not allow right shifts on signed integers.

Example Model

To generate code with right shifts on signed integers, create the following model. In this model, a signal of **Data type** `int16` feeds into a Shift Arithmetic block. In the Shift Arithmetic Block Parameters dialog box, the **Bits to shift** > **Direction** parameter is set to `Right`. The **Bits to shift** > **Number** parameter is set to `3`.



Generate Code with Right Shifts on Signed Integers

1. Open the Model Configuration Parameters dialog box and select the **Code Style** tab. The **Allow right shifts on signed integers** parameter is on by default.
2. Generate code for the model.

In the `rightshift.c` file, the `rightshift_step` function contains this code:

```
rightshift_Y.Out1 = (int16_T)(rightshift_U.In1 >> 3);
```

The signed integer `rightshift_U.In1` is shifted three bits to the right.

Generate Code That Does Not Allow Right Shifts on Signed Integers

1. Open the Model Configuration Parameters dialog box and select the **Code Style** tab. Clear the **Allow right shifts on signed integers** parameter.
2. Generate code for the model.

In the `rightshift.c` file, the `rightshift_step` function contains this code:

```
rightshift_Y.Out1 = (int16_T)asr_s32(rightshift_U.In1, 3U);
```

When you clear the **Allow right shifts on signed integers** parameter, the generated code contains a function call instead of a right shift on a signed integer. The function `asr_s32` contains this code:

```
int32_T asr_s32(int32_T u, uint32_T n)
{
  int32_T y;
  if (u >= 0) {
    y = (int32_T)((uint32_T)u >> n);
  } else {
    y = -(int32_T)((uint32_T)-(u + 1) >> n) - 1;
  }

  return y;
}
```

The `asr_s32` function casts a signed integer to an unsigned integer, and then right shifts the unsigned integer.

For more information on the **Allow right shifts on signed integers** parameter, see "Allow right shifts on signed integers".

Control Indentation Style in Generated Code

For code indentation, you can set the following parameters:

- "Indent style" controls the placement of braces in generated code.

- "Indent size" controls the number of characters per indent level in generated code (2–8 characters).

You can set **Indent style** to `K&R` or `Allman` style.

K&R

K&R stands for Kernighan and Ritchie. Each function has the opening and closing brace on its own line at the same level of indentation as the function header. Code within the function is indented according to the **Indent size**.

For blocks within a function, opening braces are on the same line as the control statement. Closing braces are on a new line at the same level of indentation as the control statement. Code within the block is indented according to the **Indent size**.

For example, here is generated code with the **Indent style** set to `K&R` with an **Indent size** of `2`:

```
void rt_OneStep(void)
{
  static boolean_T OverrunFlag = 0;
  if (OverrunFlag) {
    rtmSetErrorStatus(rtwdemo_counter_M, "Overrun");
    return;
  }

  OverrunFlag = TRUE;
  rtwdemo_counter_step();
  OverrunFlag = FALSE;
}
```

Allman

Each function has the opening and closing brace on its own line at the same level of indentation as the function header. Code within the function is indented according to the **Indent size**.

For blocks within a function, opening and closing braces for control statements are on a new line at the same level of indentation as the control statement. This is the key difference between `K&R` and `Allman` styles. Code within the block is indented according to the **Indent size**.

For example, here is generated code with the **Indent style** set to `Allman` with an **Indent size** of `4`:

```
void rt_OneStep(void)
{
    static boolean_T OverrunFlag = 0;
    if (OverrunFlag)
    {
        rtmSetErrorStatus(rtwdemo_counter_M, "Overrun");
        return;
    }

    OverrunFlag = TRUE;
    rtwdemo_counter_step();
    OverrunFlag = FALSE;
}
```

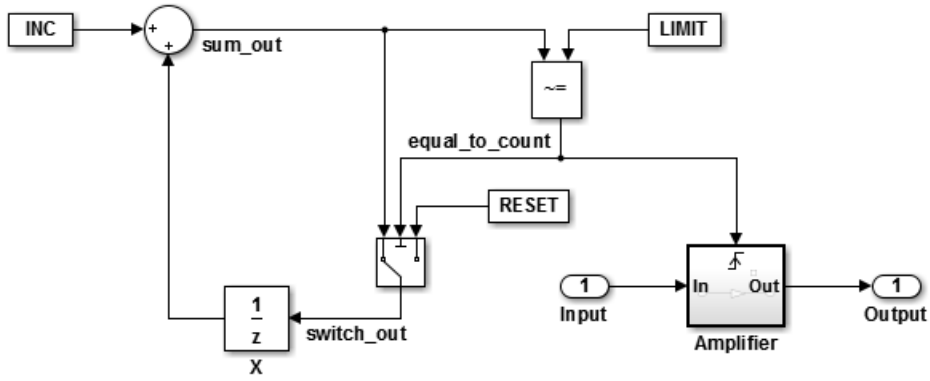
Control Cast Expressions in Generated Code

You can choose how the code generator specifies data type casts in the generated code. In the Configuration Parameters dialog box, select **Code Generation > Code Style**. From the **Casting modes** drop-down list, three parameter options control how the code generator casts data types.

- `Nominal` instructs the code generator to generate code that has minimal data type casting. When you do not have special data type information requirements, choose `Nominal`.
- `Standards Compliant` instructs the code generator to cast data types to conform to MISRA standards when it generates code. The MISRA data type casting eliminates common MISRA standard violations, including address arithmetic and assignment. It reduces 10.1, 10.2, 10.3, and 10.4 violations.

  For more information, see “MISRA C Guidelines” on page 12-5.
- `Explicit` instructs the code generator to cast data type values explicitly when it generates code. You can see how a value is stored, which tells you how much memory space the code uses for the variable. The data type informs you how much precision is possible in calculations involving the variable.

Open the example model `rtwdemo_rtwecintro`.



Enable Nominal Casting Mode and Generate Code

When you choose `Nominal` casting mode, the code generator does not create data type casts for variables in the generated code.

1. On the **Code Generation** > **Code Style** pane, from the **Casting modes** drop-down list, select `Nominal`.
2. On the **Code Generation** > **Report** pane, select **Create code generation report**.
3. On the **Code Generation** pane, select **Generate code only**.
4. Click **Apply**.
5. In the model window, press **Ctrl+B** to generate code.
6. In the Code Generation report left pane, click `rtwdemo_rtwecintro.c` to see the code.

```
/* Model step function */
void rtwdemo_rtwecintro_step(void)
{
  boolean_T rtb_equal_to_count;

  /* Sum: 'XRootX/Sum' incorporates:
   *  Constant: 'XRootX/INC'
   *  UnitDelay: 'XRootX/X'
   */
  rtDWork.X++;

  /* RelationalOperator: 'XRootX/RelOpt' incorporates:
   *  Constant: 'XRootX/LIMIT'
   */
```

```
  rtb_equal_to_count = (rtDWork.X != 16);

  /* Outputs for Triggered SubSystem: 'XRootX/Amplifier' incorporates:
   *  TriggerPort: 'XS1X/Trigger'
   */
  if (rtb_equal_to_count && (rtPrevZCSigState.Amplifier_Trig_ZCE != POS_ZCSIG))
  {
    /* Outport: 'XRootX/Output' incorporates:
     *  Gain: 'XS1X/Gain'
     *  Inport: 'XRootX/Input'
     */
    rtY.Output = rtU.Input << 1;
  }

  rtPrevZCSigState.Amplifier_Trig_ZCE = (uint8_T)(rtb_equal_to_count ? (int32_T)
    POS_ZCSIG : (int32_T)ZERO_ZCSIG);

  /* End of Outputs for SubSystem: 'XRootX/Amplifier' */

  /* Switch: 'XRootX/Switch' */
  if (!rtb_equal_to_count) {
    /* Update for UnitDelay: 'XRootX/X' incorporates:
     *  Constant: 'XRootX/RESET'
     */
    rtDWork.X = OU;
  }

  /* End of Switch: 'XRootX/Switch' */
}
```

Enable Standards Compliant Casting Mode and Generate Code

When you choose `Standards Compliant` casting mode, the code generator creates MISRA standards compliant data type casts for variables in the generated code.

1. On the **Code Style** pane, from the **Casting modes** drop-down list, select `Standards Compliant`.
2. On the **Code Generation** pane, click **Apply**.
3. In the model window, press **Ctrl+B** to generate code.
4. In the Code Generation report left pane, click `rtwdemo_rtwecintro.c` to see the code.

```
void rtwdemo_rtwecintro_step(void)
```

```
{
  boolean_T rtb_equal_to_count;

  /* Sum: '<Root>/Sum' incorporates:
   *  Constant: '<Root>/INC'
   *  UnitDelay: '<Root>/X'
   */
  rtDWork.X++;

  /* RelationalOperator: '<Root>/RelOpt' incorporates:
   *  Constant: '<Root>/LIMIT'
   */
  rtb_equal_to_count = (boolean_T)(int32_T)((int32_T)rtDWork.X != (int32_T)16);

  /* Outputs for Triggered SubSystem: '<Root>/Amplifier' incorporates:
   *  TriggerPort: '<S1>/Trigger'
   */
  if (((int32_T)rtb_equal_to_count) && (rtPrevZCSigState.Amplifier_Trig_ZCE !=
       POS_ZCSIG)) {
    /* Outport: '<Root>/Output' incorporates:
     *  Gain: '<S1>/Gain'
     *  Inport: '<Root>/Input'
     */
    rtY.Output = (int32_T)(uint32_T)((uint32_T)rtU.Input << (uint32_T)(int8_T)1);
  }

  rtPrevZCSigState.Amplifier_Trig_ZCE = (uint8_T)(int32_T)(rtb_equal_to_count ?
    (int32_T)(uint8_T)POS_ZCSIG : (int32_T)(uint8_T)ZERO_ZCSIG);

  /* End of Outputs for SubSystem: '<Root>/Amplifier' */

  /* Switch: '<Root>/Switch' */
  if (!rtb_equal_to_count) {
    /* Update for UnitDelay: '<Root>/X' incorporates:
     *  Constant: '<Root>/RESET'
     */
    rtDWork.X = OU;
  }

  /* End of Switch: '<Root>/Switch' */
}
```

Enable Explicit Casting Mode and Generate Code

When you choose `Explicit` casting mode, the code generator creates explicit data type casts for variables in the generated code.

1. On the **Code Style** pane, from the **Casting modes** drop-down list, select `Explicit`.
2. On the **Code Generation** pane, click **Apply**.
3. In the model window, press **Ctrl+B** to generate code.
4. In the Code Generation report left pane, click `rtwdemo_rtwecintro.c` to see the code.

```
/* Model step function */
void rtwdemo_rtwecintro_step(void)
{
  boolean_T rtb_equal_to_count;

  /* Sum: '<Root>/Sum' incorporates:
   *  Constant: '<Root>/INC'
   *  UnitDelay: '<Root>/X'
   */
  rtDWork.X = (uint8_T)(1U + (uint32_T)(int32_T)rtDWork.X);

  /* RelationalOperator: '<Root>/RelOpt' incorporates:
   *  Constant: '<Root>/LIMIT'
   */
  rtb_equal_to_count = (boolean_T)((int32_T)rtDWork.X != 16);

  /* Outputs for Triggered SubSystem: '<Root>/Amplifier' incorporates:
   *  TriggerPort: '<S1>/Trigger'
   */
  if (((int32_T)rtb_equal_to_count) && ((int32_T)((int32_T)
        rtPrevZCSigState.Amplifier_Trig_ZCE != (int32_T)POS_ZCSIG))) {
    /* Outport: '<Root>/Output' incorporates:
     *  Gain: '<S1>/Gain'
     *  Inport: '<Root>/Input'
     */
    rtY.Output = rtU.Input << 1;
  }

  rtPrevZCSigState.Amplifier_Trig_ZCE = (uint8_T)(rtb_equal_to_count ? (int32_T)
     POS_ZCSIG : (int32_T)ZERO_ZCSIG);

  /* End of Outputs for SubSystem: '<Root>/Amplifier' */
```

```
  /* Switch: '<Root>/Switch' */
  if (!(int32_T)rtb_equal_to_count) {
    /* Update for UnitDelay: '<Root>/X' incorporates:
     *  Constant: '<Root>/RESET'
     */
    rtDWork.X = 0U;
  }

  /* End of Switch: '<Root>/Switch' */
}
```

Related Examples

- "Conform to Coding Standards by Replacing and Renaming Data Types" on page 21-22

More About

- "Model Configuration Parameters: Code Generation Code Style"

Customize Code Organization and Format

In this section...

Custom file processing (CFP) tools allow you to customize the organization and formatting of your generated code. With these tools, you can:

- Generate a source (`.c` or `.cpp`) or header (`.h`) file. Using a *custom file processing template* (CFP template), you can control how code emits to the standard generated model files (for example, *`model`*`.c` or `.cpp`, *`model`*`.h`) or generate files that are independent of model code.
- Organize generated code into sections (such as includes, `typedef`s, functions, and more). Your CFP template can emit code (for example, functions), directives (such as `#define` or `#include` statements), or comments into each section.
- Generate custom *file banners* (comment sections) at the start and end of generated code files and custom *function banners* that precede functions in the generated code.
- Generate code to call model functions, such as *`model`*`_initialize`, *`model`*`_step`, and so on.
- Generate code to read and write model inputs and outputs.
- Generate a main program module.
- Obtain information about the model and the generated files from the model.

Custom File Processing Components

The custom file processing features are based on the following interrelated components:

- *Code generation template* (CGT) files: a CGT file defines the top-level organization and formatting of generated code. See "Code Generation Template (CGT) Files" on page 36-57.
- The *code template API*: a high-level Target Language Compiler (TLC) API that provides functions with which you can organize code into named sections and subsections of generated source and header files. The code template API also provides utilities that return information about generated files, generate standard model calls, and perform other functions. See "Code Template API Summary" on page 36-79.

- *Custom file processing (CFP) templates*: a CFP template is a TLC file that manages the process of custom code generation. A CFP template assembles code to be generated into buffers. A CFP template also calls the code template API to emit the buffered code into specified sections of generated source and header files. A CFP template interacts with a CGT file, which defines the ordering of major sections of the generated code. See "Custom File Processing (CFP) Templates" on page 36-63.

To use CFP templates, you must understand TLC programming, for more information, see "Target Language Compiler" (Simulink Coder).

Custom File Processing Configuration

Customize generated code by specifying code and data templates on the **Code Generation > Templates** pane:

Goal	Action
Specify a template that defines the top-level organization and formatting of generated source code (`.c` or `.cpp`) files	Enter a code generation template (CGT) file for the **Source file (*.c) template** parameter.
Specify a template that defines the top-level organization and formatting of generated header (`.h`) files	Enter a CGT file for the **Header file (*.h) template** parameter. This template file can be the same template file that you specify for **Source file (.c) template**. If you use the same template file, source and header files contain identical banners. The default template is *`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. For more information, see "Custom File Processing (CFP) Templates" on page 36-63.
Generate a model-specific example main program module	Select **Generate an example main program**. For more information, see "Generate a Standalone Program" on page 49-2.

Note: Place the template files that you specify on the MATLAB path.

Specify Templates For Code Generation

To use custom file processing features, create CGT files and CFP templates. These files are based on default templates provided by the code generation software. Once you have created your templates, you must integrate them into the code generation process.

Select and edit CGT files and CFP templates, and specify their use in the code generation process in the **Code Generation** > **Templates** pane of a model configuration set. The following figure shows options configured for their defaults.

The options related to custom file processing are:

- The **Source file (.c) template** field in the **Code templates** and **Data templates** sections. This field specifies the name of a CGT file to use when generating source (`.c` or `.cpp`) files. You must place this file on the MATLAB path.
- The **Header file (.h) template** field in the **Code templates** and **Data templates** sections. This field specifies the name of a CGT file to use when generating header (`.h`) files. You must place this file on the MATLAB path.

  By default, the template for both source and header files is matlabroot/toolbox/rtw/targets/ecoder/ert_code_template.cgt.
- The **File customization template** edit field in the **Custom templates** section. This field specifies the name of a CFP template file to use when generating code files. You must place this file on the MATLAB path. The default CFP template is matlabroot/toolbox/rtw/targets/ecoder/example_file_process.tlc.

In each of these fields, click **Browse** to navigate to and select an existing CFP template or CGT file. Click **Edit** to open the specified file into the MATLAB editor where you can customize it.

Code Generation Template (CGT) Files

Code Generation Template (CGT) files define the top-level organization and formatting of generated source code and header files. CGT files have the following applications:

- Generation of custom banners (comments sections) in code files. See "Generate Custom File and Function Banners" on page 36-82.
- Generation of custom code using a CFP template requires a CGT file. To use CFP templates, you must understand the CGT file structure. In many cases, however, you can use the default CGT file without modifying it.

Default CGT file

The code generation software provides a default CGT file, matlabroot/toolbox/rtw/targets/ecoder/ert_code_template.cgt. Base your custom CGT files on the default file.

CGT File Structure

A CGT file consists of one required section and four optional sections:

Code Insertion Section

(Required) This section contains tokens that define an ordered partitioning of the generated code into a number of sections (such as `Includes` and `Defines` sections). Tokens have the form of:

```
%<SectionName>
```

For example,

```
%<Includes>
```

The code generation software defines a minimal set of required tokens. These tokens generate C or C++ source or header code. They are *built-in* tokens (see "Built-In Tokens and Sections" on page 36-58). You can also define custom tokens and custom sections.

Each token functions as a placeholder for a corresponding section of generated code. The ordering of the tokens defines the order in which the corresponding sections appear in the generated code. If you do not include a token, then the corresponding section is not

generated. To generate code into a given section, explicitly call the code template API from a CFP template, as described in "Custom File Processing (CFP) Templates" on page 36-63.

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 36-60.

In the code insertion section, you can also insert C or C++ comments between tokens. Such comments emit directly into the generated code.

File Banner Section

(Optional) This section contains comments and tokens you use in generating a custom file banner.

Function Banner Section

(Optional) This section contains comments and tokens for use in generating a custom function banner.

Shared Utility Function Banner Section

(Optional) This section contains comments and tokens for use in generating a custom shared utility function banner.

File Trailer Section

(Optional) This section contains comments for use in generating a custom trailer banner.

For more information on these sections, see "Generate Custom File and Function Banners" on page 36-82.

Built-In Tokens and Sections

The following code extract shows the required code insertion section of the default CGT file with the required built-in tokens.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Code insertion section (required)
%%   These are required tokens. You can insert comments and other tokens in
%% between them, but do not change their order or remove them.
%%
```

```
%<Includes>
%<Defines>
%<Types>
%<Enums>
%<Definitions>
%<Declarations>
%<Functions>
```

Note the following requirements for customizing a CGT file:

- Do not remove required built-in tokens.
- Built-in tokens must appear in the order shown because each successive section has dependencies on previous sections.
- Only one token per line.
- Do not repeat tokens.
- You can add custom tokens and comments to the code insertion section as long as you do not violate the previous requirements.

Note: If you modify a CGT file and then rebuild your model, the code generation process does not force a top model build. To regenerate the code, see "Force Regeneration of Top Model Code" (Simulink Coder).

The following table summarizes the built-in tokens and corresponding section names, and describes the code sections.

Built-In CGT Tokens and Corresponding Code Sections

Token and Section Name	Description
`Includes`	`#include` directives section
`Defines`	`#define` directives section
`Types`	`typedef` section.`Typedefs` can depend on a previously defined type
`Enums`	Enumerated types section
`Definitions`	Data definitions (for example, `double x = 3.0;`)
`Declarations`	Data declarations (for example, `extern double x;`)
`Functions`	C or C++ functions

Subsections

You can define one or more named subsections for any section. Some of the built-in sections have predefined subsections summarized in table Subsections Defined for Built-In Sections.

Note: Sections and subsections emit to the source or header file in the order listed in the CGT file.

Using the custom section feature, you can define additional sections. See "Generate a Custom Section" on page 36-72.

Subsections Defined for Built-In Sections

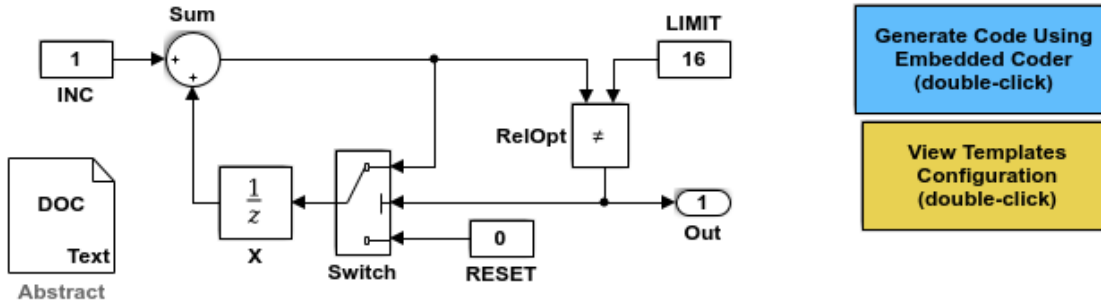
Section	Subsections	Subsection Description
`Includes`	N/A	
`Defines`	N/A	
`Types`	`IntrinsicTypes`	Intrinsic `typedef` section. Intrinsic types depend only on intrinsic C or C++ types.
`Types`	`PrimitiveTypedefs`	Primitive `typedef` section. Primitive `typedef`s depend only on intrinsic C or C++ types and on `typedef`s previously defined in the `IntrinsicTypes` section.
`Types`	`UserTop`	You can place any type of code in this section, including code that has dependencies on the previous sections.
`Types`	`Typedefs`	`typedef` section. `Typedef`s can depend on previously defined types
`Enums`	N/A	
`Definitions`	N/A	
`Declarations`	N/A	
`Functions`		C or C++ functions
`Functions`	`CompilerErrors`	`#error` directives
`Functions`	`CompilerWarnings`	`#warning` directives

Section	Subsections	Subsection Description
`Functions`	`Documentation`	Documentation (comment) section
`Functions`	`UserBottom`	You can place any code in this section.

Format Generated Code Files Using Templates

This example shows how to use code generation templates to add custom code banners, rearrange data and functions, and insert additional code segments and documentation into generated code files.

```
model='rtwdemo_codetemplate';
open_system(model)
```



<S:Note>This Simulink annotation maps to the code template %<Note> symbol.

Description
This model extracts text from the Model Description, a DOC block, and a Simulink annotation. The Simulink annotation maps to the code template %<Note> symbol. Many symbols are available that extract information from a model automatically. Refer to Module Packaging Features in the Embedded Coder documentation to see a complete list.

Many organizations employ coding standards that include consistent file layout and elaboration. Embedded Coder provides extensible code generation templates for formatting generated code. The layout and format of the code template file controls the output of the generated code.

Code generation templates (.cgt files) allow you to add custom code banners, rearrange data and functions, and insert additional code segments and documentation into generated code files. Code generation templates are picked up automatically from the MATLAB path and employed during the code generation process. For this model, the code generation template "rtwdemocodetemplate.cgt" is used.

Instructions
1. Double-click the yellow View Templates Configuration button to view the templates configuration.
2. Click the Edit button to the right of the "rtwdemocodetemplate.cgt" template, and inspect the code template.
3. Double-click the blue button in the upper right to generate code. An HTML report appears automatically.
4. Open rtwdemo_codetemplate.c to inspect the generated code for this model.


```
rtwdemoclean;
close_system(model,0)
```

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 "Generate Source and Header Files with a Custom File Processing (CFP) Template" on page 36-67.
- TLC files supporting generation of single-rate and multirate main program modules (see "Customizing Main Program Module Generation" on page 36-71).

Once you have created a CFP template, you must integrate it into the code generation process, using the **File customization template** edit field. See "Specify Templates For Code Generation" on page 36-56.

Custom File Processing (CFP) Template Structure

A custom file processing (CFP) template imposes a simple structure on the code generation process. The template, a code generation template (CGT) file, partitions the code generated for each file into a number of sections. These sections are summarized in Built-In CGT Tokens and Corresponding Code Sections and Subsections Defined for Built-In Sections.

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.

Determine the `section` argument as follows:

- If Subsections Defined for Built-In Sections does not define subsections for a given section, use the section name as the `section` argument.
- If Subsections Defined for Built-In Sections 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 "Generate a Custom Section" on page 36-72).

- `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 legality or syntax checking is not performed on the custom code within each section.

See "Generate Source and Header Files with a Custom File Processing (CFP) Template" on page 36-67, for typical usage examples.

Change the Organization of a Generated File

The files created during code generation are organized according to the general code generation template. This template has the filename `ert_code_template.cgt`, and is specified by default in **Code Generation** > **Templates** pane of the Configuration Parameters dialog box.

The following fragment shows the `rtwdemo_basicsc.c` file header that is generated using this default template:

```
/*
* File: rtwdemo_basicsc.c
*
* Code generated for Simulink model 'rtwdemo_basicsc'.
*
* Model version                  : 1.299
* Simulink Coder version         : 8.11 (R2017a) 01-Aug-2016
* C/C++ source code generated on : Fri Aug 19 12:45:59 2016
*
* Target selection: ert.tlc
* Embedded hardware selection: Intel->x86-64 (Windows64)
* Code generation objectives: Unspecified
* Validation result: Not run
*/
```

You can change the organization of generated files using code templates and data templates. Code templates organize the files that contain functions, primarily. Data templates organize the files that contain identifiers. In this procedure, you organize the generated files, using the supplied code and data templates:

1. Display the active **Templates** configuration parameters.
2. In the **Code templates** section of the **Templates** pane, type `code_c_template.cgt` into the **Source file (*.c) templates** text box.
3. Type `code_h_template.cgt` into the **Header file (*.h) templates** text box.
4. In the **Data templates** section, type `data_c_template.cgt` into the **Source file (*.c) templates** text box.
5. Type `data_h_template.cgt` into the **Header file (*.h) templates** text box, and click **Apply**.
6. In the model window, press **Ctrl+B**. Now the files are organized using the templates you specified. For example, the `rtwdemo_basicsc.c` file header now is organized like this:

```
/**
 ***************************************************************************
 **  FILE INFORMATION:
 **  Filename:           rtwdemo_basicsc.c
 **  File Creation Date: 19-Aug-2016
 **
 **  ABSTRACT:
 **
 **
 **  NOTES:
 **
 **
 **  MODEL INFORMATION:
 **  Model Name:         rtwdemo_basicsc
 **  Model Description:  Specifying Storage Class Within a Diagram

   This model shows how to define data storage class as part of
   the diagram.
 **  Model Version:      1.299
 **  Model Author:       The MathWorks, Inc. - Mon Nov 27 14:36:56 2000
 **
 **  MODIFICATION HISTORY:
 **  Model at Code Generation: user - Fri Aug 19 12:47:36 2016
 **
 **  Last Saved Modification:  The MathWorks, Inc. - Sat Aug 06 14:37:49 2016
 **
 **
 ***************************************************************************
 **/
```

Generate Source and Header Files with a Custom File Processing (CFP) Template

In this section...

This example shows you the process of generating a simple source (`.c` or `.cpp`) and header (`.h`) file using the example CFP template. Then, it examines the template and the code generated by the template.

The example CFP template, matlabroot/toolbox/rtw/targets/ecoder/example_file_process.tlc, demonstrates some of the capabilities of the code template API, including

- Generation of simple source (`.c` or `.cpp`) and header (`.h`) files
- Use of buffers to generate file sections for includes, functions, and so on
- Generation of includes, defines, into the standard generated files (for example, *`model`*`.h`)
- Generation of a main program module

Generate Code with a CFP Template

This section sets up a CFP template and configures a model to use the template in code generation. The template generates (in addition to the standard model files) a source file (`timestwo.c` or `.cpp`) and a header file (`timestwo.h`).

Follow the steps below to become acquainted with the use of CFP templates:

1. Copy the example CFP template, matlabroot/toolbox/rtw/targets/ecoder/example_file_process.tlc, to a folder outside of the MATLAB folder structure (that is, not under *`matlabroot`*). If the folder is not on the MATLAB path or the TLC path, then add it to the MATLAB path. It is good practice to locate the CFP template in the same folder as your system target file, which is on the TLC path.

2. Rename the copied `example_file_process.tlc` to `test_example_file_process.tlc`.
3. Open `test_example_file_process.tlc` into the MATLAB editor.
4. Uncomment the following line:

   ```
   %%  %assign ERTCustomFileTest = TLC_TRUE
   ```

   It now reads:

   ```
     %assign ERTCustomFileTest = TLC_TRUE
   ```

   If `ERTCustomFileTest` is not assigned as shown, the CFP template is ignored in code generation.
5. Save your changes to the file. Keep `test_example_file_process.tlc` open, so you can refer to it later.
6. Open the rtwdemo_udt model.
7. Open the Simulink Model Explorer. Select the active configuration set of the model, and open the **Code Generation** pane of the active configuration set.
8. On the **Templates** tab, in the **File customization template** field, specify `test_example_file_process.tlc`. This is the file you previously edited and is now the specified CFP template for your model.
9. On the **General** tab, select the **Generate code only** check box.
10. Click **Apply**.
11. In the model window, press **Ctrl+B**. During code generation, notice the following message in the **Diagnostic Viewer**:

    ```
    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.
12. The `rtwdemo_udt` model is configured to generate an HTML code generation report. After code generation is complete, view the report.

    Notice that the **Generated Code** list contains the following files:

    - Under **Main file**, `ert_main.c`.
    - Under **Other files**, `timestwo.c` and `timestwo.h`.

The files were generated by the CFP template. The next section examines the template to learn how this was done.

13 Keep the model, the code generation report, and the `test_example_file_process.tlc` file open so you can refer to them in the next section.

Analysis of the Example CFP Template and Generated Code

This section examines excerpts from `test_example_file_process.tlc` and some of the code it generates. Refer to the comments in matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc while reading the following discussion.

Generating Code Files

Source (`.c` or `.cpp`) and header (`.h`) files are created by calling `LibCreateSourceFile`, as in the following excerpts:

```
%assign cFile = LibCreateSourceFile("Source", "Custom", "timestwo")
...
%assign hFile = LibCreateSourceFile("Header", "Custom", "timestwo")
```

Subsequent code refers to the files by the file reference returned from `LibCreateSourceFile`.

File Sections and Buffers

The code template API lets you partition the code generated to each file into sections, tagged as `Definitions`, `Includes`, `Functions`, `Banner`, and so on. You can append code to each section as many times as required. This technique gives you a great deal of flexibility in the formatting of your custom code files.

Subsections Defined for Built-In Sections describes the available file sections and their order in the generated file.

For each section of a generated file, use `%openfile` and `%closefile` to store the text for that section in temporary buffers. Then, to write (append) the buffer contents to a file section, call `LibSetSourceFileSection`, passing in the desired section tag and file reference. For example, the following code uses two buffers (`typesBuf` and `tmpBuf`) to generate two sections (tagged `"Includes"` and `"Functions"`) of the source file `timestwo.c` or `.cpp` (referenced as `cFile`):

```
%openfile typesBuf

#include "rtwtypes.h"

%closefile typesBuf

%<LibSetSourceFileSection(cFile,"Includes",typesBuf)>

 %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 "rtwtypes.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 name for the *`model`*`.h` file. It then obtains a file reference and generates a definition in the `Defines` section of *`model`*`.h`:

```
%% Add a #define to the model's public header file model.h

%assign pubName = LibGetMdlPubHdrBaseName()
%assign modelH  = LibCreateSourceFile("Header", "Simulink", pubName)

%openfile tmpBuf
```

```
 #define ACCELERATION 9.81

 %closefile tmpBuf

%<LibSetSourceFileSection(modelH,"Defines",tmpBuf)>
```

Examine the generated `rtwdemo_udt.h` file to see the generated `#define` directive.

Customizing Main Program Module Generation

Normally, the ERT target determines whether and how to generate an `ert_main.c` or `.cpp` module based on the settings of the **Generate an example main program** and **Target operating system** options on the **Templates** pane of the Configuration Parameters dialog box. You can use a CFP template to override the normal behavior and generate a main program module customized for your target environment.

To support generation of main program modules, two TLC files are provided:

- `bareboard_srmain.tlc`: TLC code to generate an example single-rate main program module for a bareboard target environment. Code is generated by a single TLC function, `FcnSingleTaskingMain`.
- `bareboard_mrmain.tlc`: TLC code to generate a multirate main program module for a bareboard target environment. Code is generated by a single TLC function, `FcnMultiTaskingMain`.

In the example CFP template file matlabroot/toolbox/rtw/targets/ecoder/example_file_process.tlc, the following code generates either a single- or multitasking `ert_main.c` or `.cpp` module. The logic depends on information obtained from the code template API calls `LibIsSingleRateModel` and `LibIsSingleTasking`:

```
%% Create a simple main.  Files are located in MATLAB/rtw/c/tlc/mw.

 %if LibIsSingleRateModel() || LibIsSingleTasking()
   %include "bareboard_srmain.tlc"
   %<FcnSingleTaskingMain()>
 %else
   %include "bareboard_mrmain.tlc"
   %<FcnMultiTaskingMain()>
 %endif
```

Note that `bareboard_srmain.tlc` and `bareboard_mrmain.tlc` use the code template API to generate `ert_main.c` or `.cpp`.

When generating your own main program module, you disable the default generation of `ert_main.c` or `.cpp`. The TLC variable `GenerateSampleERTMain` controls generation

of `ert_main.c` or `.cpp`. You can directly force this variable to `TLC_FALSE`. The examples `bareboard_mrmain.tlc` and `bareboard_srmain.tlc` use this technique, as shown in the following excerpt from `bareboard_srmain.tlc`.

```
%if GenerateSampleERTMain
    %assign CompiledModel.GenerateSampleERTMain = TLC_FALSE
    %warning Overriding example ert_main.c!
%endif
```

Alternatively, you can implement a `SelectCallback` function for your target. A `SelectCallback` function is a MATLAB function that is triggered during model loading, and also when the user selects a target with the System Target File browser. Your `SelectCallback` function should deselect and disable the **Generate an example main program** option. This prevents the TLC variable `GenerateSampleERTMain` from being set to TLC_TRUE.

See the "rtwgensettings Structure" (Simulink Coder) section 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 49-4 and "Guidelines for Modifying rt_OneStep" on page 49-9 for further information.

Generate 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 subsections. Custom sections must be associated with one of the built-in sections: `Includes`, `Defines`, `Types`, `Enums`, `Definitions`, `Declarations`, or `Functions`. To create custom sections, you must

- Add a custom token to the code insertion section of your CGT file.
- In your CFP file:
  - Assemble code to be generated to the custom section into a buffer.
  - Declare an association between the custom section and a built-in section, with the code template API function `LibAddSourceFileCustomSection`.
  - Emit code to the custom section with the code template API function `LibSetSourceFileCustomSection`.

The following code examples illustrate the addition of a custom token, `Myincludes`, to a CGT file, and the subsequent association of the custom section `Myincludes` with the built-in section `Includes` in a CFP file.

Note: If you have not already created custom CGT and CFP files for your model, copy the default template files matlabroot/toolbox/rtw/targets/ecoder/ert_code_template.cgt and matlabroot/toolbox/rtw/targets/ecoder/example_file_process.tlc to a work folder that is outside the MATLAB folder structure but on the MATLAB or TLC path, rename them (for example, add the prefix `test_` to each file), and update the **Templates** pane of the Configuration Parameters dialog box to reference them.

First, add the token `Myincludes` to the code insertion section of your CGT file. For example:

```
%<Includes>
%<Myincludes>
%<Defines>
%<Types>
%<Enums>
%<Definitions>
%<Declarations>
%<Functions>
```

Next, in the CFP file, add code to generate `include` directives into a buffer. For example, in your copy of the example CFP file, you could insert the following section between the `Includes` section and the `Create a simple main` section:

```
%% Add a custom section to the model's C file model.c

%openfile tmpBuf
#include "moretables1.h"
```

```
#include "moretables2.h"
%closefile tmpBuf

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

The `LibAddSourceFileCustomSection` function call declares an association between the built-in section `Includes` and the custom section `Myincludes`. `Myincludes` is a subsection of `Includes`. The `LibSetSourceFileCustomSection` function call directs the code in the `tmpBuf` buffer to the `Myincludes` section of the generated file. `LibSetSourceFileCustomSection` is syntactically identical to `LibSetSourceFileSection`.

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

```
#include "rtwdemo_udt.h"
#include "rtwdemo_udt_private.h"

/* #include "mytables.h" */
#include "moretables1.h"
#include "moretables2.h"
```

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

Custom Tokens

Custom tokens are automatically translated to TLC syntax as a part of the build process. To escape a token, that is to prepare it for normal TLC expansion, use the '!' character. For example, the token `%<!TokenName>` is expanded to `%<TokenName>` by the template conversion program. You can specify valid TLC code, including TLC function calls: `%<!MyTLCFcn()>`.

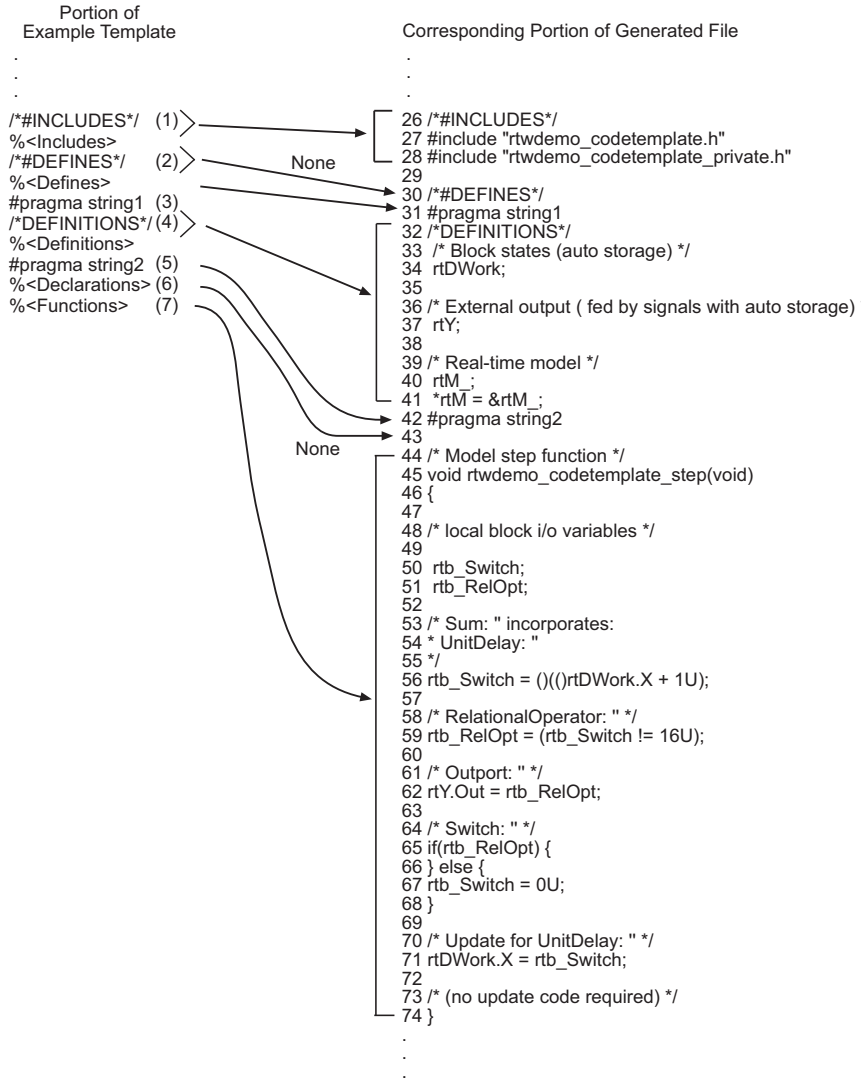
Comparison of a Template and Its Generated File

This figure shows part of a user-modified custom file processing (CFP) template and the resulting generated code. The figure illustrates how you can use a template to:

- Define what code the code generation software should add to the generated file
- Control the location of code in the file
- Optionally insert comments in the generated file

Notice `%<Includes>`, for example, on the template. The term `Includes` is a symbol name. A percent sign and brackets (`%< >`) must enclose every symbol name. You can add the desired symbol name (within the `%< >` delimiter) at a particular location in the template. This is how you control where the code generator places an item in the generated file.

Template and Generated File



Mapping Template Specification to Code Generation

This part of the template...		Generates in the file...		Explanation
		Line	Description	
(1)	`/*#INCLUDES*/` `%<Includes>`	26–28	An `/*#INCLUDES*/` comment, followed by `#include` statements	The code generator adds the C/C++ comment as a header, and then interprets the `%<Includes>` template symbol to list the required `#include` statements in the file. This code is first in this section of the file because the template entries are first.
(2)	`/*#DEFINES*/` `%<Defines>`	30	A `/*#DEFINES*/` comment, but no `#define` statements	Next, the code generator places the comment as a header for `#define` statements, but the file does not need `#define`. No code is added.
(3)	`#pragma string1`	31	`#pragma` statements	While the code generator requires `%<>` delimiters for template symbols, it can also interpret C/C++ statements in the template without delimiters. In this case, the generator adds the specified statements to the code, following the order in which the statements appear in the template.
(5)	`#pragma string2`	42		
(4)	`/*DEFINITIONS*/` `%<Definitions>`	32–41	`/*DEFINITIONS*/` comment, followed by definitions	The code generator places the comment and definitions in the file between the `#pragma` statements, according to the order in the template. It also inserts comments (lines 33 and 36) that are preset in the model's Configuration Parameters dialog box.
(6)	`%<Declarations>`	43	No declarations	The file needs no declarations, so the code generator does not generate declarations for this file. The template does not have

This part of the template...		Generates in the file...		Explanation
		Line	Description	
				a comment to provide a header. Line 43 is left blank.
(7)	`%<Functions>`	44–74	Functions	Finally, the code generator adds functions from the model, plus comments that are preset in the Configuration Parameters dialog box. But it adds no comments as a header for the functions, because the template does not have one. This code is last because the template entry is last.

For a list of template symbols and the rules for using them, see “Template Symbol Groups” on page 36-90, “Template Symbols” on page 36-93, and “Rules for Modifying or Creating a Template” on page 36-96. To set comment options, from the **Simulation** menu, select **Model Configuration Parameters**. On the Configuration Parameters dialog box, select the **Code Generation > Comments** pane. For details, see “Configure Code Comments” (Simulink Coder).

Code Template API Summary

Code Template API Functions summarizes the code template API. See the source code in matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc for detailed information on the arguments, return values, and operation of these calls.

Code Template API Functions

Function	Description
`LibClearFileSectionContents`	Clears a file section with custom values before writing file to disk.
`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.
`LibGetFileRecordName`	Returns a model file name (including the path) without the extension.
`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 36-63).
`LibIndentSourceFile`	Indents a file (from within the TLC environment).
`LibCallModelInitialize`	Returns code for calling the model's *`model`*`_initialize` function (valid for ERT only).

Function	Description
`LibCallModelStep`	Returns code for calling the model's *`model`*`_step` function (valid for ERT only).
`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 (without an 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 single-tasking execution.
`LibWriteModelInput`	Returns the code to write to a particular root input (that is, a model inport block). (valid for ERT only).
`LibWriteModelOutput`	Returns the code to write to a particular root output (that is, a model outport block). (valid for ERT only).

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 folder 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`.

Generate Custom File and Function Banners

Using code generation template (CGT) files, you can specify custom file banners and function banners for the generated code files. File banners are comment sections in the header and trailer sections of a generated file. Function banners are comment sections for each function in the generated code. Use these banners to add a company copyright statement, specify a special version symbol for your configuration management system, remove time stamps, and for many other purposes. These banners can contain characters, which propagate to the generated code.

To specify banners, create a custom CGT file with customized banner sections. The build process creates an executable TLC file from the CGT file. The code generation process then invokes the TLC file.

You do not need to be familiar with TLC programming to generate custom banners. You can modify example files that are supplied with the ERT target.

Note Prior releases supported direct use of customized TLC files as banner templates. You specified these with the **Source file (.c) banner template** and **Header file (.h) banner template** options of the ERT target. You can still use a custom TLC file banner templates, however, you can now use CGT files instead.

ERT template options on the **Code Generation > Templates** pane of a configuration set, in the **Code templates** section, support banner generation.

The options for function and file banner generation are:

- "Code templates: Source file (*.c) template": CGT file to use when generating source (`.c` or `.cpp`) files. Place this file on the MATLAB path.
- "Code templates: Header file (*.h) template": CGT file to use when generating header (`.h`) files. You must place this file on the MATLAB path. This file can be the same template specified in the **Code templates: Source file (*.c) template** field, in which case identical banners are generated in source and header files.

  By default, the template for both source and header files is matlabroot/toolbox/rtw/targets/ecoder/ert_code_template.cgt.
- In each of these fields, click **Browse** to navigate to and select an existing CGT file for use as a template. Click **Edit** to open the specified file into the MATLAB editor, where you can customize it.

Create a Custom File and Function Banner Template

To customize a CGT file for custom banner generation, make a local copy of the default code template and edit it, as follows:

1. Activate the configuration set that you want to work with.
2. Open the **Code Generation** pane of the active configuration set.
3. Click the **Templates** tab.
4. By default, the code template specified in the **Code templates: Source file (*.c) template** and **Code templates: Header file (*.h) template** fields is matlabroot/toolbox/rtw/targets/ecoder/ert_code_template.cgt.
5. If you want to use a different template as your starting point, click **Browse** to locate and select a CGT file.
6. Click **Edit** button to open the CGT file into the MATLAB editor.
7. Save a local copy of the CGT file. Store the copy in a folder that is outside of the MATLAB folder structure, but on the MATLAB path. If required, add the folder to the MATLAB path.
8. If you intend to use the CGT file with a custom target, locate the CGT file in a folder under your target root folder.
9. Rename your local copy of the CGT file. When you rename the CGT file, update the associated **Code templates: Source file (*.c) template** or **Code templates: Header file (*.h) template** field to match the new file name.
10. Edit and customize the local copy of the CGT file for banner generation, using the information provided in "Customize a Code Generation Template (CGT) File for File and Function Banner Generation" on page 36-84.
11. Save your changes to the CGT file.
12. Click **Apply** to update the configuration set.
13. Save your model.
14. Generate code. Examine the generated source and header files to confirm that they contain the banners specified by the template or templates.

Customize a Code Generation Template (CGT) File for File and Function Banner Generation

This section describes how to edit a CGT file for custom file and function banner generation. For a description of CGT files, see "Code Generation Template (CGT) Files" on page 36-57.

Components of the File and Function Banner Sections in the CGT file

In a CGT file, you can modify the following sections: file banner, function banner, shared utility function banner, and file trailer. Each section is defined by open and close tags. The tags specific to each section are shown in the following table.

CGT File Section	Open Tag	Close Tag
File Banner	`<FileBanner>`	`</FileBanner>`
Function Banner	`<FunctionBanner>`	`</FunctionBanner>`
Shared-utility Banner	`<SharedUtilityBanner>`	`</SharedUtilityBanner>`
File Trailer	`<FileTrailer>`	`</FileTrailer>`

You can customize your banners by including tokens and comments between the open and close tag for each section. Tokens are typically TLC variables, for example `<ModelVersion>`, which are replaced with values in the generated code.

Note: Including C comment indicators, '/*' or a '*/', in the contents of your banner might introduce an error in the generated code.

An open tag includes tag attributes. Enclose the value of the attribute in double quotes. The attributes available for an open tag are:

- `width`: specifies the width of the file or function banner comments in the generated code. The default value is 80.
- `style`: specifies the boundary for the file or function banner comments in the generated code.

The open tag syntax is as follows:

```
<OpenTag style = "style_value" width = "num_width">
```

Note: If the **Configuration Parameters > Code Generation > Language** parameter is set to `C++`, to select a comment style that uses C comment notation (`/*...*/`), you must also set the **Configuration Parameters > All Parameters > Comment style** parameter to `Multi-line`. For more information, see "Specify Comment Style" on page 36-14.

The built-in style options for the `style` attribute are:

- `classic`

```
/* single line comments */
/*
 * multiple line comments
 * second line
 */
```

- `classic_cpp`

```
// single line comments
//
// multiple line comments
// second line
//
```

- `box`

```
/*********************************************************/
/* banner contents                                       */
/*********************************************************/
```

- `box_cpp`

```
///////////////////////////////////////////////////////////
// banner contents                                       //
///////////////////////////////////////////////////////////
```

- `open_box`

```
/*********************************************************
 * banner contents
 *********************************************************/
```

- `open_box_cpp`

```
///////////////////////////////////////////////////////////
// banner contents
```

```
////////////////////////////////////////////////////////////
```

- `doxygen`

```
/** single line comments */

/**
 * multiple line comments
 * second line
 */
```

- `doxygen_cpp`

```
/// single line comments

///
/// multiple line comments
/// second line
///
```

- `doxygen_qt`

```
/*! single line comments */

/*!
 * multiple line comments
 * second line
 */
```

- `doxygen_qt_cpp`

```
//! single line comments

//!
//! multiple line comments
//! second line
//!
```

File Banner

This section contains comments and tokens for use in generating a custom file banner. The file banner precedes C or C++ code generated by the model. If you omit the file banner section from the CGT file, then no file banner emits to the generated code.

Note: If you customize your file banner, the software does not emit the customized banner for the file `const_params.c`.

The following section is the file banner section provided with the default CGT file, matlabroot/toolbox/rtw/targets/ecoder/ert_code_template.cgt.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Custom file banner section (optional)
%%
<FileBanner style="classic">
File: %<FileName>

Code generated for Simulink model %<ModelName>.

Model version                        : %<ModelVersion>
Simulink Coder version               : %<RTWFileVersion>
TLC version                          : %<TLCVersion>
C/C++ source code generated on       : %<SourceGeneratedOn>
%<CodeGenSettings>
</FileBanner>
```

Summary of Tokens for File Banner Generation

`FileName`	Name of the generated file (for example, `"rtwdemo_udt.c"`).
`FileType`	Either `"source"` or `"header"`. Designates whether generated file is a `.c` or `.cpp` file or an `.h` file.
`FileTag`	Given file names `file.c` or `.cpp` and `file.h`; the file tags are `"file_c"` and `"file_h"`, respectively.
`ModelName`	Name of generating model.
`ModelVersion`	Version number of model.
`RTWFileVersion`	Version number of *`model`*`.rtw` file.
`RTWFileGeneratedOn`	Timestamp of *`model`*`.rtw` file.
`TLCVersion`	Version of Target Language Compiler.
`SourceGeneratedOn`	Timestamp of generated file.
`CodeGenSettings`	Code generation settings for model: target language, target selection, production hardware selection, test hardware selection, code generation objectives (in priority order), and Code Generation Advisor validation result.

Function Banner

This section contains comments and tokens for use in generating a custom function banner. The function banner precedes C or C++ function generated during the build

process. If you omit the function banner section from the CGT file, the default function banner emits to the generated code. The following section is the default function banner section provided with the default CGT file, matlabroot/toolbox/rtw/targets/ecoder/ert_code_template.cgt.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Custom function banner section (optional)
%%   Customize function banners by using the following predefined tokens:
%% %<ModelName>, %<FunctionName>, %<FunctionDescription>, %<Arguments>,
%% %<ReturnType>, %<GeneratedFor>, %<BlockDescription>.
%%
<FunctionBanner style="classic">
%<FunctionDescription>
%<BlockDescription>
</FunctionBanner>
```

Summary of Tokens for Function Banner Generation

`FunctionName`	Name of function
`Arguments`	List of function arguments
`ReturnType`	Return type of function
`ModelName`	Name of generating model
`FunctionDescription`	Short abstract about the function
`GeneratedFor`	Full block path for the generated function
`BlockDescription`	User input from the **Block Description** parameter of the block properties dialog box. `BlockDescription` contains an optional token attribute, `style`. The only valid value for`style` is `content_only`, which is case-sensitive and enclosed in double quotes. Use the `content_only` style when you want to include only the block description content that you entered in the block parameter dialog. The syntax for the token attribute `style` is: `%<BlockDescription style = ”content_only”>`

Shared Utility Function Banner

The shared utility function banner section contains comments and tokens for use in generating a custom shared utility function banner. The shared utility function banner precedes C or C++ shared utility function generated during the build process. If you omit the shared utility function banner section from the CGT file, the default shared utility function banner emits to the generated code. The following section is the default shared

utility function banner section provided with the default CGT file, matlabroot/toolbox/rtw/targets/ecoder/ert_code_template.cgt.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Custom shared utility function banner section (optional)
%%   Customize banners for functions generated in shared location by using the
%% following predefined tokens: %<FunctionName>, %<FunctionDescription>,
%% %<Arguments>, %<ReturnType>.
%%
<SharedUtilityBanner style="classic">
%<FunctionDescription>
</SharedUtilityBanner>
```

Summary of Tokens for Shared Utility Function Banner Generation

`FunctionName`	Name of function
`Arguments`	List of function arguments
`ReturnType`	Return type of function
`FunctionDescription`	Short abstract about function

File Trailer

The file trailer section contains comments for generating a custom file trailer. The file trailer follows C or C++ code generated from the model. If you omit the file trailer section from the CGT file, no file trailer emits to the generated code. The following section is the default file trailer provided in the default CGT file.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Custom file trailer section (optional)
%%
<FileTrailer style="classic">
File trailer for generated code.

[EOF]
</FileTrailer>
```

Tokens available for the file banner are available for the file trailer. See Summary of Tokens for File Banner Generation.

Template Symbols and Rules

In this section...

Introduction

"Template Symbol Groups" on page 36-90 and "Template Symbols" on page 36-93 describe custom file processing (CFP) template symbols and rules for using them. The location of a symbol in one of the supplied template files (`code_c_template.cgt`, `code_h_template.cgt`, `data_c_template.cgt`, or `data_h_template.cgt`) determines where the items associated with that symbol are located in the corresponding generated file. "Template Symbol Groups" on page 36-90 identifies the symbol groups, starting with the parent ("Base") group, followed by the children of each parent. "Template Symbols" on page 36-93 lists the symbols alphabetically.

Note: If you are using custom CGT sections, for files generated to the `_sharedutils` folder, you can only use symbol names in the Base symbol group.

Template Symbol Groups

Symbol Group	Symbol Names in This Group

Base (Parents)	`Declarations`
	`Defines`
	`Definitions`
	`Documentation`
	`Enums`
	`Functions`

Symbol Group	Symbol Names in This Group
	`Includes` `Types`

Declarations	`ExternalCalibrationLookup1D`
	`ExternalCalibrationLookup2D`
	`ExternalCalibrationScalar`
	`ExternalVariableScalar`

Defines	`LocalDefines` `LocalMacros`

Definitions	`FilescopeCalibrationLookup1D`
	`FilescopeCalibrationLookup2D`
	`FilescopeCalibrationScalar`
	`FilescopeVariableScalar`
	`GlobalCalibrationLookup1D`
	`GlobalCalibrationLookup2D`
	`GlobalCalibrationScalar`
	`GlobalVariableScalar`

Symbol Group	Symbol Names in This Group

Documentation	`Abstract`
	`Banner`
	`Created`
	`Creator`
	`Date`
	`Description`
	`FileName`
	`History`
	`LastModifiedDate`
	`LastModifiedBy`
	`ModelName`
	`ModelVersion`
	`ModifiedBy`
	`ModifiedComment`
	`ModifiedHistory`

	`Notes` `ToolVersion`
Functions	`CFunctionCode`
Types	This parent has no children.

Template Symbols

Symbol Name*	Symbol Group	Symbol Scope	Symbol Description (What the symbol puts in the generated file)
`Abstract`	Documentation	N/A	User-supplied description of the model or file. Placed in the generated file based on the Stateflow note, Simulink annotation, or DocBlock on the model.**
`Banner`	Documentation	N/A	Comments located near top of the file. Contains information that includes model and software versions, and date file was generated.
`CFunctionCode`	Functions	File	C/C++ functions. Must be at the bottom of the template.
`Created`	Documentation	N/A	Date when model was created. From **Created on** field on Model Properties dialog box.
`Creator`	Documentation	N/A	User who created model. From **Created by** field on Model Properties dialog box.
`Date`	Documentation	N/A	Date file was generated. Taken from computer clock.
`Declarations`	Base		Data declaration of a signal or parameter. For example, `extern real_T globalvar;`.
`Defines`	Base	File	Required `#defines` of `.h` files.
`Definitions`	Base	File	Data definitions of signals or parameters.
`Description`	Documentation	N/A	Description of model. From **Model description** field on Model Properties dialog box.**

Symbol Name*	Symbol Group	Symbol Scope	Symbol Description (What the symbol puts in the generated file)
`Documentation`	Base	N/A	Comments about how to interpret the generated files.
`Enums`	Base	File	Enumerated data type definitions.
`ExternalCalibrationLookup1D`	Declarations	External	***
`ExternalCalibrationLookup2D`	Declarations	External	***
`ExternalCalibrationScalar`	Declarations	External	***
`ExternalVariableScalar`	Declarations	External	***
`FileName`	Documentation	N/A	Name of the generated file.
`FilescopeCalibrationLookup1I`	Definitions	File	***
`FilescopeCalibrationLookup2I`	Definitions	File	***
`FilescopeCalibrationScalar`	Definitions	File	***
`FilescopeVariableScalar`	Definitions	File	***
`Functions`	Base	File	Generated function code.
`GlobalCalibrationLookup1D`	Definitions	Global	***
`GlobalCalibrationLookup2D`	Definitions	Global	***
`GlobalCalibrationScalar`	Definitions	Global	***
`GlobalVariableScalar`	Definitions	Global	***
`History`	Documentation	N/A	User-supplied revision history of the generated files. Placed in the generated file based on the Stateflow note, Simulink annotation, or DocBlock on the model.**
`Includes`	Base	File	`#include` preprocessor directives.
`LastModifiedDate`	Documentation	N/A	Date when model was last saved. From **Last saved on** field on Model Properties dialog box.

Symbol Name*	Symbol Group	Symbol Scope	Symbol Description (What the symbol puts in the generated file)
`LastModifiedBy`	Documentation	N/A	User who last saved model. From **Last saved by** field on Model Properties dialog box.
`LocalDefines`	Defines	File	`#define` preprocessor directives from code-generation data objects.
`LocalMacros`	Defines	File	C/C++ macros local to the file.
`ModelName`	Documentation	N/A	Name of the model.
`ModelVersion`	Documentation	N/A	Version number of the Simulink model. From **Model version** field on Model Properties dialog box.
`ModifiedBy`	Documentation	N/A	Name of user who last modified the model.
`ModifiedComment`	Documentation	N/A	Comment user enters in the **Modified Comment** field on the Log Change dialog box. For more information, see "Log Comments History" (Simulink).
`ModifiedHistory`	Documentation	N/A	Text from **Model history** field on Model Properties dialog box.**
`Notes`	Documentation	N/A	User-supplied miscellaneous notes about the model or generated files. Placed in the generated file based on the Stateflow note, Simulink annotation, or DocBlock on the model.**
`ToolVersion`	Documentation	N/A	A list of the versions of the toolboxes used in generating the code.
`Types`	Base		Data types of generated code.

* Symbol names must be enclosed between `%< >`. For example, `%<Functions>`.

** This symbol can be used to add a comment to the generated files. See "Add Global Comments" on page 36-8. The code generator places the comment in each generated file whose template has this symbol name. The code generator places the comment at the location that corresponds to where the symbol name is located in the template file.

*** The description can be deduced from the symbol name. For example, `GlobalCalibrationScalar` is a symbol that identifies a scalar. It contains data of global scope that you can calibrate .

Rules for Modifying or Creating a Template

The following are the rules for creating a MPF template. "Comparison of a Template and Its Generated File" on page 36-75 illustrates several of these rules.

1. Place a symbol on a template within the `%< >` delimiter. For example, the symbol named `Includes` should look like this on a template: `%<Includes>`. *Note that symbol names are case sensitive.*
2. Place a symbol on a template where desired. Its location on the template determines where the item associated with this symbol is located in the generated file. If no item is associated with it, the symbol is ignored.
3. Place a C/C++ statement outside of the `%< >` delimiter, and on a different line than a `%< >` delimiter, for that statement to appear in the generated file. For example, `#pragma message ("my text")` in the template results in `#pragma message ("my text")` at the corresponding location in the generated file. Note that the statement must be compatible with your C/C++ compiler.
4. Use the `.cgt` extension for every template filename. ("`cgt`" stands for *c*ode *g*eneration *t*emplate.)
5. Note that `%% $Revision: 1.1.4.10.4.1 $` appears at the top of the MathWorks supplied templates. This is for internal MathWorks use only. It does not need to be placed on a user-defined template and does not show in a generated file.
6. Place a comment on the template between `/* */` as in standard ANSI C[6]. This results in `/*comment*/` on the generated file.
7. Each MPF template must have all of the Base group symbols, in predefined order. They are listed in "Template Symbol Groups" on page 36-90. Each symbol in the Base group is a parent. For example, `Declarations` is a parent symbol.

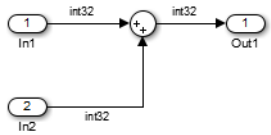
6. ANSI is a registered trademark of the American National Standards Institute, Inc.

8. Each symbol in a non-Base group is a child. For example, `LocalMacros` is a child.
9. Except for Documentation children, children must be placed after their parent, before the next parent, and before the `Functions` symbol.
10. Documentation children can be located before or after their parent in any order anywhere in the template.
11. If a non-Documentation child is missing from the template, the code generator places the information associated with this child at its parent location in the generated file.
12. If a Documentation child is missing from the template, the code generator omits the information associated with that child from the generated file.

Annotate Code for Justifying Polyspace Checks

With the Polyspace Code Prover product you can apply Polyspace verification to Embedded Coder generated code. The software detects run-time errors in the generated code and helps you to locate and fix model faults.

Polyspace might highlight overflows for certain operations that are legitimate because of the way the code generator implements these operations. Consider the following model and the corresponding generated code.




```
32 /* Sum: '<Root>/Sum' incorporates:
33   *  Inport: '<Root>/In1'
34   *  Inport: '<Root>/In2'
35   */
36  qY_0 = sat_add_U.In1 + sat_add_U.In2;
37  if ((sat_add_U.In1 < 0) && ((sat_add_U.In2 < 0) && (qY_0 >= 0))) {
38    qY_0 = MIN_int32_T;
39  } else {
40   if ((sat_add_U.In1 > 0) && ((sat_add_U.In2 > 0) && (qY_0 <= 0))) {
41      qY_0 = MAX_int32_T;
42    }
43  }
```

The code generator recognizes that the largest built-in data type is 32-bit. It is not possible to saturate the results of the additions and subtractions using `MIN_INT32` and `MAX_INT32` and a bigger single-word integer data type. Instead the software detects the results overflow and the direction of the overflow, and saturates the result.

If you do not provide justification for the addition operator on line 36, Polyspace verification generates an orange check that indicates a potential overflow. The verification does not take into account the saturation function of lines 37 to 43. In addition, the trace-back functionality of Polyspace Code Prover does not identify the reason for the orange check.

To justify overflows from operators that are legitimate, on the **Configuration Parameters > Code Generation > Comments** pane:

- Under **Overall control**, select the **Include comments** check box.
- Under **Auto generate comments**, select the **Operator annotations check box**.

The code generator annotates generated code with comments for Polyspace. For example:

```
32 /* Sum: '<Root>/Sum' incorporates:
33   *  Inport: '<Root>/In1'
34   *  Inport: '<Root>/In2'
35   */
36  qY_0 = sat_add_U.In1 +/*MW:OvOk*/ sat_add_U.In2;
```

When you run a verification using Polyspace Code Prover, the Polyspace software uses the annotations to justify the operator-related orange checks and assigns the `Not a defect` classification to the checks.

Manage Placement of Data Definitions and Declarations

In this section...

Overview of Data Placement

This chapter focuses on module packaging features (MPF) settings that are interdependent. Their combined values, along with Simulink partitioning, determine the file placement of data definitions and declarations, or *data placement*. This includes

- The number of files generated.
- Whether or not the generated files contain definitions for a model's global identifiers. And, if a definition exists, the settings determine the files in which MPF places them.
- Where MPF places global data declarations (`extern`).

The following six MPF settings are distributed among the main procedures and form an important interdependency:

- The **Data definition** field on the **Code Placement** pane of the Configuration Parameters dialog box.
- The **Data declaration** field on the **Code Placement** pane of the Configuration Parameters dialog box.
- The **Owner** field of the data object in the Model Explorer and the checkbox for **Use owner from data object for data definition placement** on the **Code Placement** pane of the Configuration Parameters dialog box. The term "ownership settings" refers to these fields together.
- The **Definition file** field of the data object on the Model Explorer.
- The **Header file** field of the data object on the Model Explorer.

- The **Memory section** field of the data object on the Model Explorer.

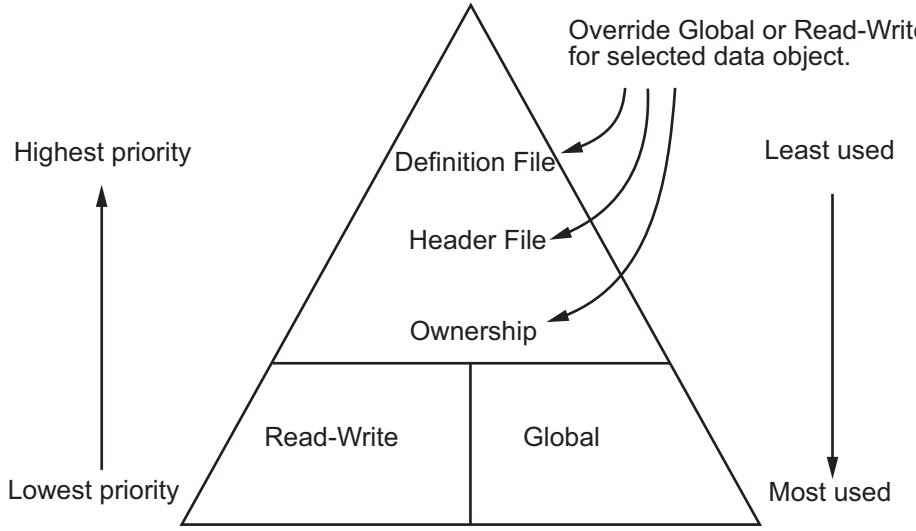
Priority and Usage

Overview

There is a priority order among interdependent MPF settings. From highest to lowest, the priorities are

- Definition File priority
- Header File priority
- Ownership priority
- Read-Write priority or Global priority

Priority order varies inversely with frequency of use, as illustrated below. For example, Definition File is highest priority but least used.



MPF Settings Priority and Usage

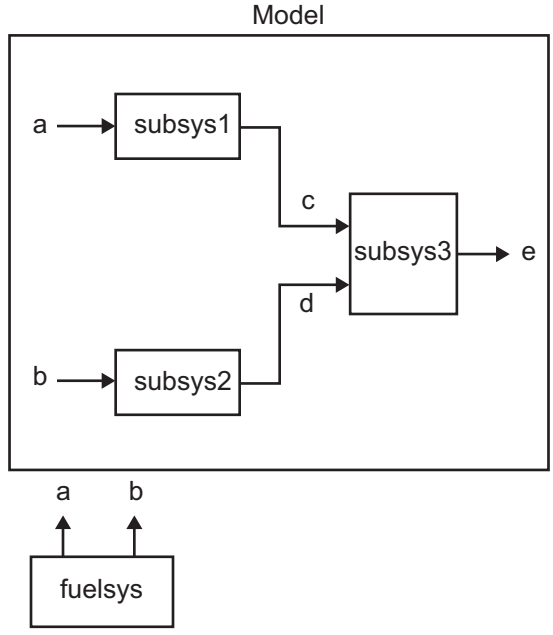
Unless they are overridden, the Read-Write and Global priorities place in the generated files all of the model's MPF-derived data objects that you selected using Data Object Wizard. (See “Create Data Objects for Code Generation with Data Object Wizard” on page 24-2 for details.) Before generating the files, you can use the higher priority Definition file, Header file, and Ownership, as desired, to override Read-Write or Global priorities for single data objects. Most users will employ Read-Write or Global, without an override. A few users, however, will want to do an override for certain data objects. We expect that those users whose applications include multiple modules will want to use the Ownership priority.

The priorities are used only for those data objects that are derived from `Simulink.Signal` and `Simulink.Parameter`, and whose custom storage classes are specified using the Custom Storage Class Designer. (For details, see “Design Custom Storage Classes and Memory Sections” on page 23-34.) Otherwise, the build process determines the data placement.

Read-Write Priority

This is the lowest priority. Consider that a model consists of one or more Simulink blocks or Stateflow diagrams. There can be subsystems within these. For the purpose of illustration, think of a model with one top-level block called `fuelsys`. You double-clicked the block and now see three subsystems labeled `subsys1`, `subsys2` and `subsys3`, as shown in the next figure. Signals `a` and `b` are outputs from the top-level block (`fuelsys`). Signal `a` is an input to `subsys1` and `b` is input to `subsys2`. Signal `c` is an output from `subsys1`. Notice the other inputs and outputs (`d` and `e`). Signals `a` through `e` have corresponding data objects.

As explained in “Data Definition and Declaration Management”, MPF provides you with the means of selecting a data object that you want defined as an identifier in the generated code. MPF also allows you to specify property values for each data object.

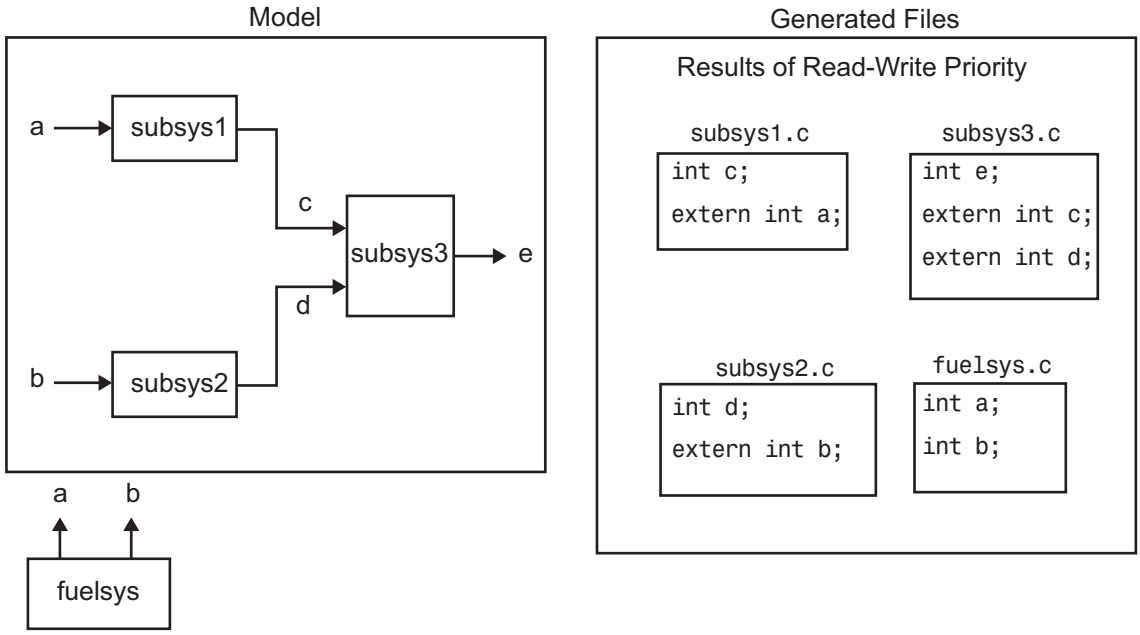


The Generated Files

We generate code for this model. As shown in the figure below, this results in a `.c` source file corresponding to each of the subsystems. (In actual applications, there could be more than one `.c` source file for a subsystem. This is based on the file partitioning previously selected for the model. But for our illustration, we only need to show one for each subsystem.) Data objects `a` through `e` have corresponding identifiers in the generated files.

A `.c` source file has one or more functions in it, depending on the internal operations (functions) of its corresponding subsystem. An identifier in a generated `.c` file has local scope when it is used only in one function of that `.c` file. An identifier has file scope when more than one function in the same `.c` file uses it. An identifier has global scope when more than one of the generated files uses it.

A subsystem's source file contains the definitions for that subsystem's data objects that have local scope or file scope. (These definitions are not shown in the figure.) But where are the definitions and declarations for data objects of global scope? These are shown in the next figure.



For the Read-Write priority, this source file contains the definitions for the subsystem's global data objects, if this is the file that first writes to the data object's address. Other files that read (use) that data object only include a reference to it. This is why this priority is called Read-Write. Since a read and a write of a file are analogous to input and output of a model's block, respectively, there is another way of saying this. The definitions of a block's global data objects are located in the corresponding generated file, if that data object is an output from that block. The declarations (`extern`) of a block's global data objects are located in the corresponding generated file, if that data object is an input to that block.

Settings for Read-Write Priority

The generated files and what they include, as just described, occur when the Read-Write priority is used. For this to be the case, the other priorities are turned off. That is,

- The **Data definition** field on the **Code Placement** pane is set to `Data defined in source file`.
- The **Data declaration** field on the **Code Placement** pane is set to `Data declared in source file`.

- The **Owner** field on the Model Explorer is blank, and the checkbox for the **Use owner from data object for data definition placement** field on the **Code Placement** pane is not checked.
- **Definition file** and **Header file** on the Model Explorer are blank.

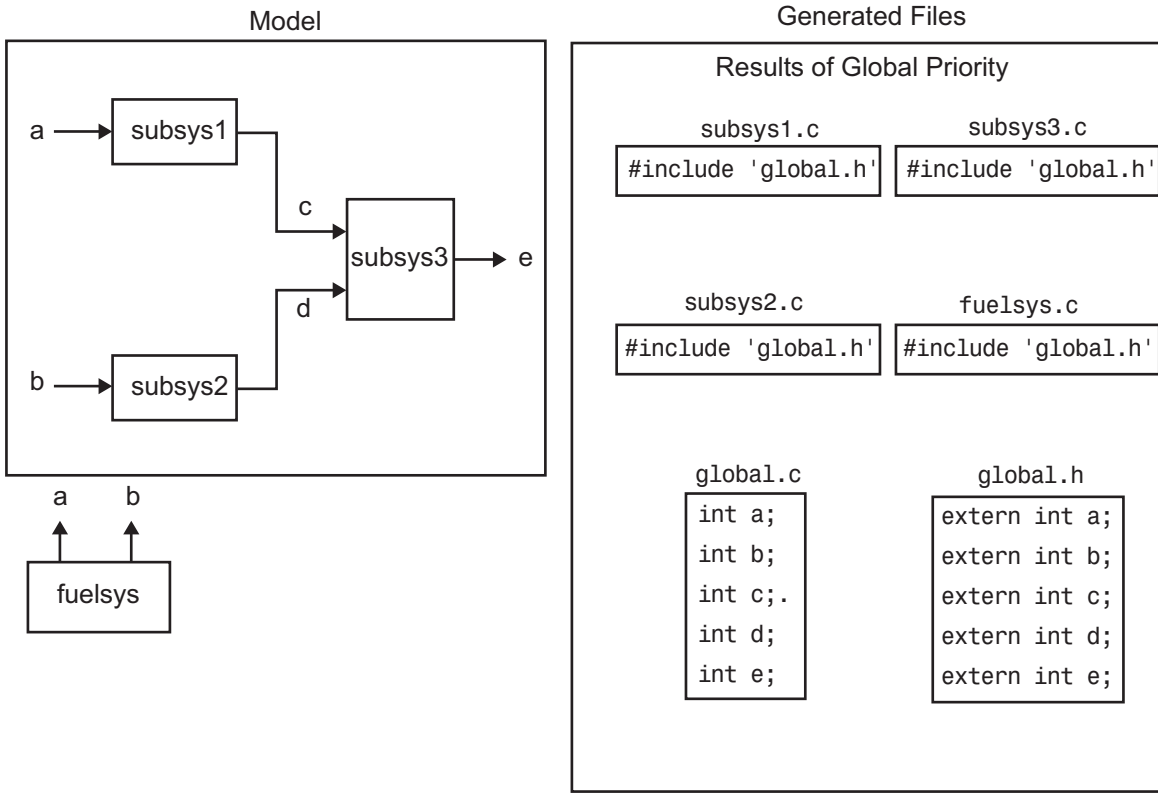
Global Priority

This has the same priority as Read-Write (the lowest) priority. The settings for this are the same as for Read-Write Priority, except

- The **Data definition** field on the **Code Placement** pane is set to `Data defined in single separate source file`.
- The **Data declaration** field on the **Code Placement** pane is set to `Data declared in single separate header file`.

The generated files that result are shown in the next figure. A subsystem's data objects of local or file scope are defined in the `.c` source file where the subsystem's functions are located (not shown). The data objects of global scope are defined in another `.c` file (called `global.c` in the figure). The declarations for the subsystem's data objects of global scope are placed in a `.h` file (called `global.h`).

For example, data objects of local and file scope for `subsys1` are defined in `subsys1.c`. Signal `c` in the model is an output of `subsys1` and an input to `subsys2`. So `c` is used by more than one subsystem and thus is a global data object. Because of the global priority, the definition for `c` (`int c;`) is in `global.c`. The declaration for `c` (`extern int c;`) is in `global.h`. Since `subsys2` uses (reads) `c`, `#include "global.h"` is in `subsys2.c`.



Definition File, Header File, and Ownership Priorities

While the Read-Write and Global priorities operate on all MPF-derived data objects that you want defined in the generated code, the remaining priorities allow you to override the Read-Write or Global priorities for one or more particular data objects. There is a high-to-low priority among these remaining priorities — Definition File, Header File, and Ownership — for a particular data object, as shown in MPF Settings Priority and Usage

Ownership Settings

Ownership settings refers to the `on` or `off` setting specified using the **Use owner from data object for data definition placement** checkbox on the **Code Placement** pane of the Configuration Parameters dialog box, and the **Owner** field of a data object in the Model Explorer. These settings control the file placement of data definition and

initialization by specifying a model that owns the data. The settings do not control what files are generated. There are four possible configurations, as shown in "Ownership Settings" on page 36-116.

Memory Section Settings

Memory sections allow you to specify storage directives for a data object. As shown in Parameter and Signal Property Values, the possible values for the **Memory section** property of a parameter or signal object are `Default`, `MemConst`, `MemVolatile` or `MemConstVolatile`.

If you specify a filename for **Definition file**, and select `Default`, `MemConst`, `MemVolatile` or `MemConstVolatile` for the **Memory section** property, the code generation software generates a `.c` file and an `.h` file. The `.c` file contains the definition for the data object with the `pragma` statement or qualifier associated with the **Memory section** selection. The `.h` file contains the declaration for the data object. The `.h` file can be included, using the preprocessor directive `#include`, in files that need to reference the data object.

You can add more memory sections. For more information, see "Design Custom Storage Classes and Memory Sections" on page 23-34 and "Control Data and Function Placement in Memory by Inserting Pragmas" on page 27-2.

Data Placement Rules

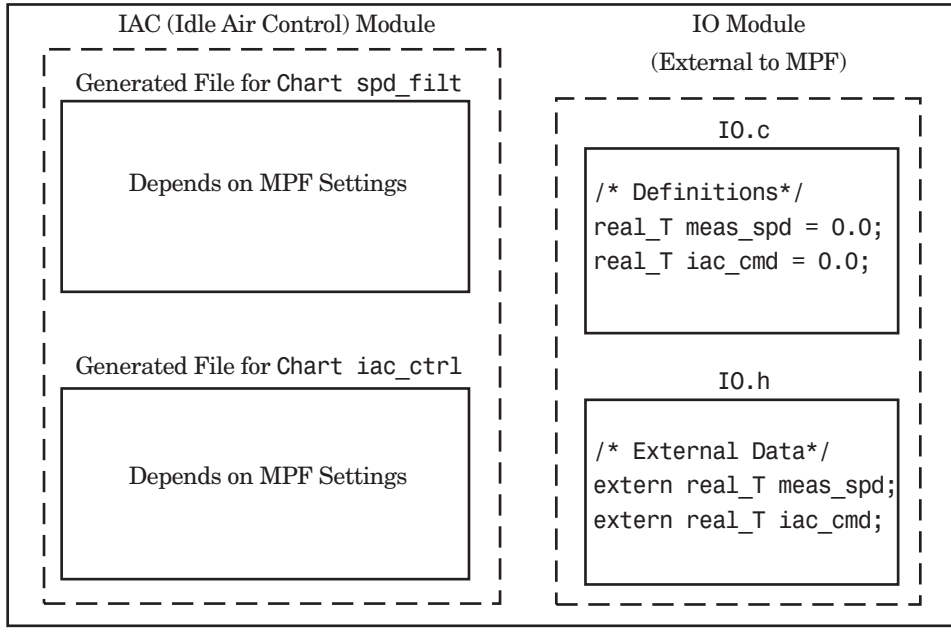
For a complete set of data placement rules in convenient tabular form, based on the priorities discussed in this chapter, see "Data Placement Rules and Results" on page 36-115.

Settings for a Data Object

Introduction

"Settings and Resulting Generated Files" on page 36-116 provides example settings for one data object of a model. Eight examples are listed so that you can see the generated files that result from a wide variety of settings. Four examples from this table are discussed below in more detail. These discussions provide information for understanding settings you might choose. For illustration purposes, the four examples assume that we are dealing with an overall system that controls engine idle speed.

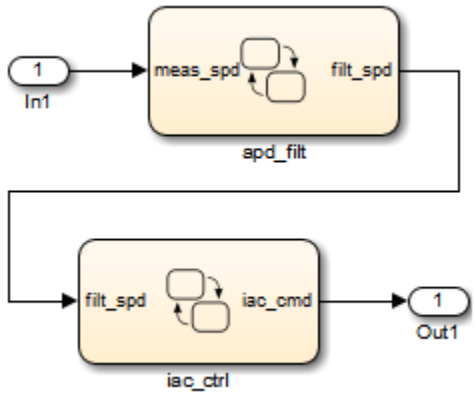
The next figure shows that the software component of this example system consists of two modules, IAC (Idle Air Control), and IO (Input-Output).



Engine Idle Speed Control System

The code in the IO module controls the system's IO hardware. Code is generated only for the IAC module. (Some other means produced the code for the IO module, such as hand-coding.) So the code in IO is external to MPF, and can illustrate legacy code. To simplify matters, the IO code contains one source file, called `IO.c`, and one header file, called `IO.h`.

The IAC module consists of two Stateflow charts, `spd_filt` and `iac_ctrl`. The `spd_filt` chart has two signals (`meas_spd`) and `filt_spd`), and one parameter (`a`). The `iac_ctrl` chart also has two signals (`filt_spd` and `iac_cmd`) and a parameter (`ref_spd`). (The parameters are not visible in the top-level charts.) One file for each chart is generated. This example system allows us to illustrate referencing from file to file within the MPF module, and model to external module. It also illustrates the case where there is no such referencing.



Proceed to the discussion of the desired example settings:

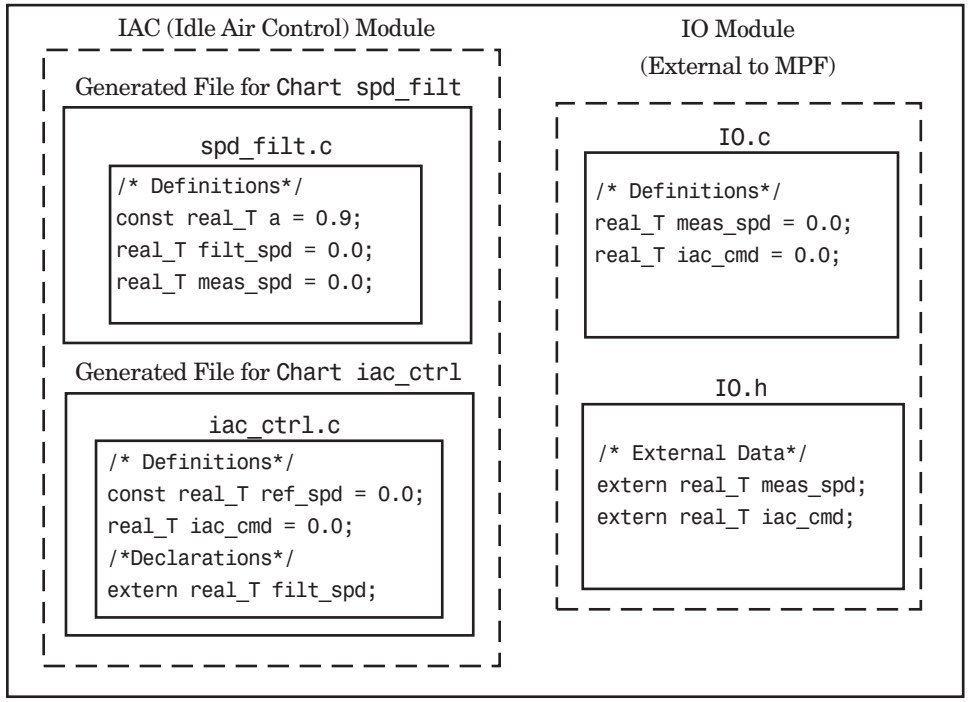
- "Read-Write" on page 36-109
- "Ownership" on page 36-111
- "Header File" on page 36-112
- "Definition File" on page 36-114

Read-Write

These settings and the generated files that result are shown as Example Settings 1 in "Settings and Resulting Generated Files" on page 36-116. As you can see from the table, this example illustrates the case in which only one `.c` source file (for each chart) is generated.

So, for the IAC model, select the following settings. Accept the `Data defined in source file` in the **Data definition** field and the `Data declared in source file` in the **Data declaration** field on the **Code Placement** pane of the Configuration

Parameters dialog box. Accept the default unchecked **Use owner from data object for data definition placement** field. Accept the default blank settings for the **Owner**, **Definition file** and **Header file** fields on the Model Explorer. For **Memory section**, accept `Default`. Now the Read-Write priority is active. Generate code. The next figure shows the results in terms of definition and declaration statements.



Engine Idle Speed Control System (Read-Write Example)

The code generator generated a `spd_filt.c` for the `spd_filt` chart and `iac_ctrl.c` for the `iac_ctrl` chart. As you can see, MPF placed definitions of data objects for the `spd_filt` chart in `spd_filt.c`. It placed definitions of data objects for the `iac_ctrl` chart in `iac_ctrl.c`.

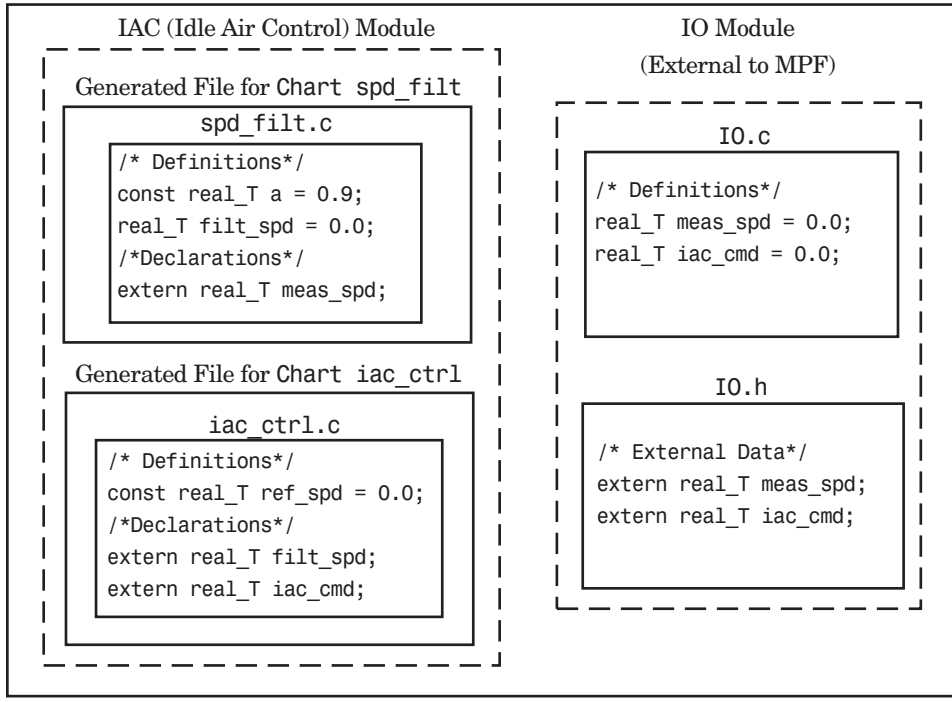
However, notice `real_T filt_spd`. This data object is defined in `spd_filt.c` and declared in `iac_ctrl.c`. That is, since the Read-Write priority is active, `filt_spd` is defined in the file that first writes to its address. And, it is declared in the file that reads (uses) it. Further, `real_T meas_spd` is defined in both `spd_filt.c` and the external `IO.c`. And, `real_T iac_cmd` is defined in both `iac_ctrl.c` and `IO.c`.

Ownership

See tables "Ownership Settings" on page 36-116 and "Settings and Resulting Generated Files" on page 36-116. In the "Read-Write" on page 36-109, there are several instances where the same data object is defined in more than one `.c` source file, and there is no declaration (`extern`) statement. This would result in compiler errors during link time. But in this example, we configure MPF Ownership rules so that linking can take place. Notice the Example Settings 2 row in "Settings and Resulting Generated Files" on page 36-116. Except for the ownership settings, assume these are the settings you made for the model in the IAC module. Since this example has no **Definition file** or **Header file** specified, now Ownership takes priority. (If you specified a **Definition file** or **Header file**, MPF ignores the ownership settings.)

On the **Code Placement** pane of the Configuration Parameters dialog box, check the box for the **Use owner from data object for data definition placement** field. Open the Model Explorer (by issuing the MATLAB command `daexplr`) and, for all data objects except `meas_spd` and `iac_cmd`, type IAC in the **Owner** field (case sensitive). Then, only for the `meas_spd` and `iac_cmd` data objects, type IO as their **Owner** (case sensitive). Generate code.

The results are shown in the next figure. Notice the `extern real_T meas_spd` statement in `spd_filt.c`, and `extern real_T iac_cmd` in `iac_ctrl.c`. MPF placed these declaration statements in the files where these data objects are used. This allows the generated source files (`spd_filt.c` and `iac_ctrl.c`) to be compiled and linked with `IO.c`.



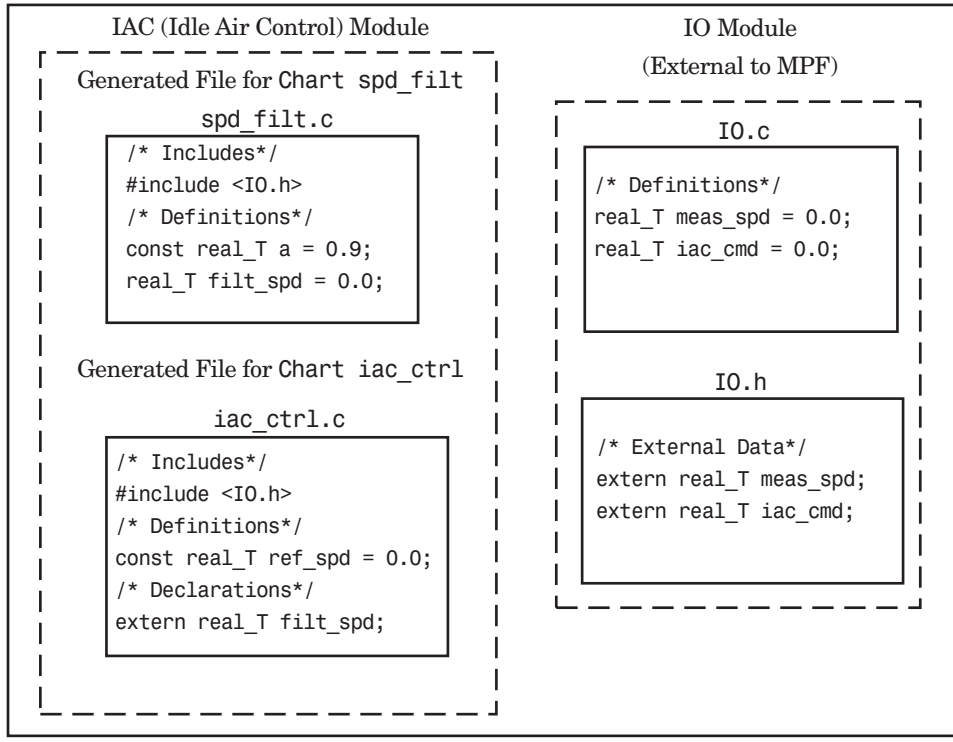
Engine Idle Speed Control System (Ownership Example)

Header File

These settings and the generated files that result are shown as Example Settings 3 in "Settings and Resulting Generated Files" on page 36-116. This example has no **Definition file** specified. If you specified a **Definition file**, MPF ignores the **Header file** setting. The focus of this example is to show how the **Header file** settings result in the linking of the two chart source files to the external IO files, shown in the next figure. (Also, ownership settings will be used to link the two chart files with each other.)

As you can see in the figure, the `meas_spd` and `iac_cmd` identifiers are defined in `IO.c` and declared in `IO.h`. Both of these identifiers are external to the generated `.c` files. You open the Model Explorer and select both the `meas_spd` and `iac_cmd` data objects. For each of these data objects, in the **Header file** field, specify `IO.h`, since this is where these two objects are declared. This setting allows the `spd_filt.c` source file to compile and link with the external `IO.c` file.

Now we configure the ownership settings. In the Model Explorer, select the `filt_spd` data object and set its **Owner** field to `IAC`. Then, on the **Code Placement** pane of the Configuration Parameters dialog box, check the box for the **Use owner from data object for data definition placement** field. Now the `spd_filt` source file links to the `iac_ctrl` source file. Generate code. See the figure below.



Engine Idle Speed Control System (Header File Example)

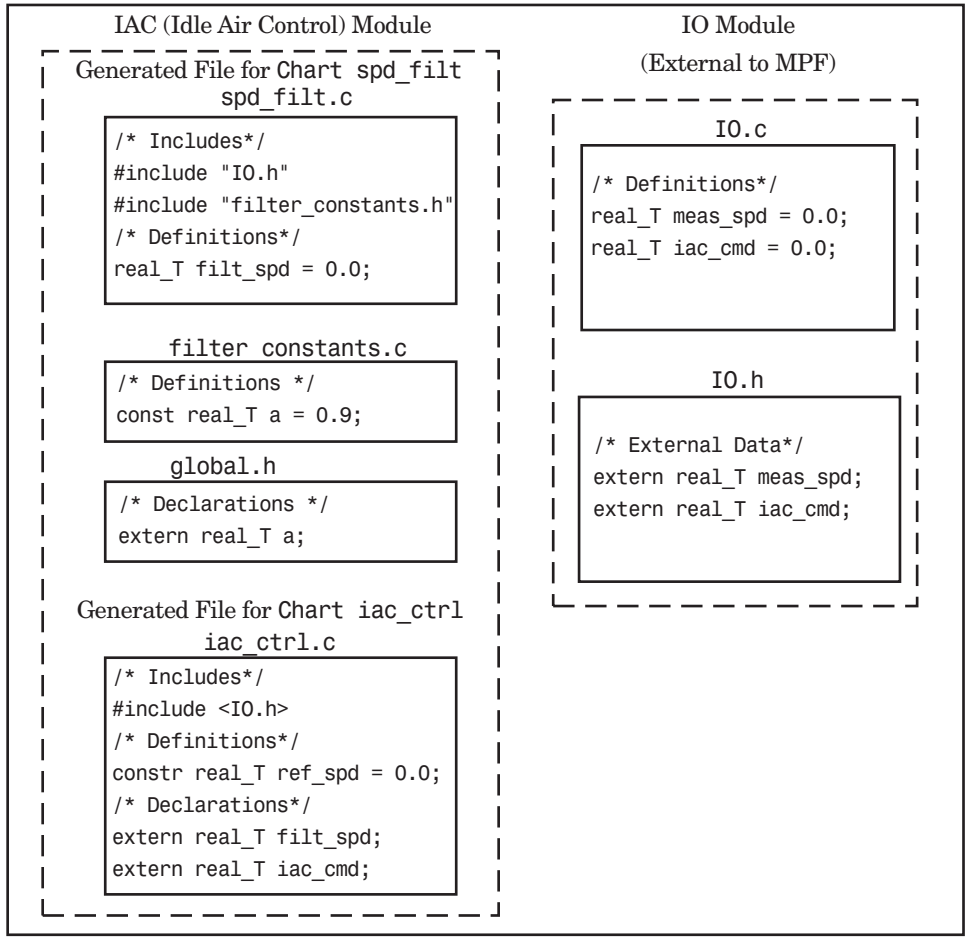
Since you specified the `IO.h` filename for the **Header file** field for the `meas_spd` and `iac_ctrl` objects, the code generator assumed that their declarations are in `IO.h`. So the code generator placed `#include IO.h` in each source file: `spd_filt.c` and `iac_ctrl.c`. So these two files will link with the external IO files. Also, due to the ownership settings that were specified, the code generator places the `real_T filt_spd = 0.0;` definition in `spd_filt.c` and declares the `filt_spd` identifier in `iac_ctrl.c` with `extern real_T iac_cmd;`. Consequently, the two source files will link together.

Definition File

These settings and the generated files that result are shown as Example Settings 4 in "Settings and Resulting Generated Files" on page 36-116. Notice that a definition filename is specified. The settings in the table only apply to the data object called `a`. You have decided that you do not want this object defined in `spd_filt.c`, the generated source file for the `spd_filt` chart. (There are many possible organizational reasons one might want an object declared in another file. It is not important for this example to specify the reason.)

For this example, assume the settings for all data objects are the same as those indicated in "Header File" on page 36-112, except for the data object `a`. The description below identifies only the differences that result from this.

Open the Model Explorer, and select data object `a`. In the **Definition file** field specify a filename. Choose `filter_constants.c`. Generate code. The results are shown in the next figure.



Engine Idle Speed Control System (Definition File Example)

The code generator generates the same files as in the “Header File” on page 36-112, and adds a new file, `filter_constants.c`. Data object `a` now is defined in `filter_constants.c`, rather than in the source file `spd_filt.c`, as it is in the example. This data object is declared with an `extern` statement in `global.h`

Data Placement Rules and Results

- “Ownership Settings” on page 36-116

- "Settings and Resulting Generated Files" on page 36-116
- "Data Placement Rules" on page 36-118

Ownership Settings

Row Number	Enable Data Ownership Checkbox	Owner Setting	Result*
1	`Off**`	Blank**	The code generator determines whether the current model defines and initializes data.
2	`Off**`	A name is specified.	The code generator determines whether the current model defines and initializes data.
3	`On`	Blank**	The code generator determines whether the current model defines and initializes data.
4	`On`	A name is specified.	The model specified in the **Owner** setting defines and initializes data.

* See also "Ownership Settings" on page 36-106.** Default.

Settings and Resulting Generated Files

	Data Defined In...	Data Declared In...	Owner-ship*	Defined File**	Header File	Generated Files
Example Settings 1 (Rd-Write Example)	Source file	Source file	Blank	Blank	Blank	`.c/.cpp` source file
Example Settings 2 (Owner- ship Example)	Source file	Source file	Name of module specified	Blank	Blank	`.c/.cpp` source file
Example Settings 3 (Header File Example)	Source file	Source file	Blank	Blank	Desired include filename specified.	`.c/.cpp` source file `.h` definition file

	Data Defined In...	Data Declared In...	Owner-ship*	Defined File**	Header File	Generated Files
Example Settings 4 (Def. File Example)	Source file	Source file	Blank	Desired definition filename specified.	Desired include filename specified.	`.c/.cpp` source file `.c/.cpp` definition file* `.h` definition file*
Example Settings 5	Single separate source file	Source file	Blank	Blank	Blank	`.c/.cpp` source file global `.c/.cpp`
Example Settings 6	Single separate source file	Single separate header file	Blank	Blank	Blank	`.c/.cpp` source file `global .c/.cpp` `global.h`
Example Settings 7	Single separate source file	Single separate header file	Name of module specified	Blank	Blank	`.c/.cpp` source file `global.c/.cppglobal.h`
Example Settings 8	Single separate source file	Single separate header file	Blank	Blank	Desired include filename specified.	`.c/.cpp` source file `global.c/.cppglobal.h` `.h` definition file

* "Blank" in ownership setting means that the check box for the **Use owner from data object for data definition placement** field on the **Code Placement** pane is `Off` and the **Owner** field on the Model Explorer is blank. "Name of module specified" can be a variety of ownership settings as defined in “Ownership Settings” on page 36-116.

** The code generator generates a definition `.c/.cpp` file for every data object for which you specified a definition filename (unless you selected `#DEFINE` for the **Memory section** field). For example, if you specify the same definition filename for all data objects, only one definition `.c/.cpp` file is generated. The code generator places declarations in *`model`*`.h` by default, unless you specify `Data declared in single separate header file` for the **Data declaration** option on the **Code Generation** > **Code Placement** pane of the Configuration Parameter dialog box. If you select that data placement option, the code generator places declarations in `global.h`. If you specify a definition filename for each data object, the code generator generates one definition `.c/.cpp` file for each data object and places declarations in *`model`*`.h` by default, unless you specify `Data declared in single separate header file` for **Data declaration**. If you select that data placement option, the code generator places declarations in `global.h`.

Note: If you generate C++ rather than C code, the `.c` files listed in the following table will be `.cpp` files.

Data Placement Rules

	Global Settings:		Override Settings for Specific Data Object:			Results in Generated Files:		
Storage Class Setting	**Data Def.**	**Data Dec.**	**Def. File**	**Owner**	**Header File**	**Where Data Def. Is**	**Where Data Dec. Is**	**Dec. Inclusion**
mpt or Simulink Noncustom Storage Classes:								
`auto`	N/A	N/A	N/A	N/A	N/A	Note 12	`model.h`	Note 1
`Exported-Global`	N/A	N/A	N/A	N/A	N/A	`model.c`	`model.h`	Note 1
`Imported-Extern`, `Imported-Extern-Pointer`	N/A	N/A	N/A	N/A	N/A	None. External	`model_pri`	Note 2
`Simulink-Global`	N/A	N/A	N/A	N/A	N/A	Note 13	`model.h`	Note 1
mpt or Simulink Custom Storage Class: Imported Data:								
`Imported-FromFile`	D/C	D/C	D/C	N/A	null	None	`model_pri`	Note 3
`Imported-FromFile`	D/C	D/C	D/C	N/A	`hdr.h`	None	`model_pri`	Note 4
Simulink Custom Storage Class: #define Data:								
`Define`	D/C	D/C	N/A	N/A	N/A	N/A	`#define`, `model.h`	Note 5
mpt Custom Storage Class: #define Data:								
`Define`	D/C	D/C	N/A	N/A	null	N/A	`#define`, `model.h`	Note 5

	Global Settings:		Override Settings for Specific Data Object:			Results in Generated Files:		
Storage Class Setting	**Data Def.**	**Data Dec.**	**Def. File**	**Owner**	**Header File**	**Where Data Def. Is**	**Where Data Dec. Is**	**Dec. Inclusion**
`Define`	D/C	D/C	N/A	N/A	`hdr.h`	N/A	`#define`, `model.h`	Note 6
mpt or Simulink Custom Storage Class: GetSet:								
`GetSet`	D/C	D/C	N/A	N/A	`hdr.h`	N/A	External `hdr.h`	Note 4
mpt or Simulink Custom Storage Class: Bitfield, Struct:								
`Bitfield`, `Struct`	D/C	D/C	N/A	N/A	N/A	`model.c`	`model.h`	Note 7
mpt or Simulink Custom Storage Class: Global, Const, ConstVolatile, Volatile:								
`Global`, `Const`, `Const-Volatile`, `Volatile`	`auto`	`auto`	null	null or locally owned	null	`model.c`	`model.h`	Note 1
`Global`, `Const`, `Const-Volatile`, `Volatile`	`src`	`auto`	null	null or locally owned	null	`src.c`	`model.h`	Note 1
`Global`, `Const`, `Const-Volatile`, `Volatile`	`sep`	`auto`	null	null or locally owned	null	`gbl.c`	`model.h`	Note 1
`Global`, `Const`, `Const-Volatile`, `Volatile`	`auto`	`src`	null	null or locally owned	null	`model.c`	`src.c`	Note 8
`Global`, `Const`, `Const-Volatile`, `Volatile`	`src`	`src`	null	null or locally owned	null	`src.c`	`src.c`	Note 8

	Global Settings:		Override Settings for Specific Data Object:			Results in Generated Files:		
Storage Class Setting	**Data Def.**	**Data Dec.**	**Def. File**	**Owner**	**Header File**	**Where Data Def. Is**	**Where Data Dec. Is**	**Dec. Inclusion**
`Global`, `Const`, `Const-Volatile`, `Volatile`	`sep`	`src`	null	null or locally owned	null	`gbl.c`	`src.c`	Note 8
`Global`, `Const`, `Const-Volatile`, `Volatile`	`auto`	`sep`	null	null or locally owned	null	`model.c`	`gbl.h`	Note 9
`Global`, `Const`, `Const-Volatile`, `Volatile`	`src`	`sep`	null	null or locally owned	null	`src.c`	`gbl.h`	Note 9
`Global`, `Const`, `Const-Volatile`, `Volatile`	`sep`	`sep`	null	null or locally owned	null	`gbl.c`	`gbl.h`	Note 9
`Global`, `Const`, `Const-Volatile`, `Volatile`	D/C	D/C	data.c	D/C	null	`data.c`	See Note 10.	Note 10
`Global`, `Const`, `Const-Volatile`, `Volatile`	D/C	D/C	data.c	D/C	`hdr.h`	`data.c`	`hdr.h`	Note 11
`Global`, `Const`, `Const-Volatile`, `Volatile`	`auto`	D/C	null	null	`hdr.h`	`model.c`	`hdr.h`	Note 11

	Global Settings:		Override Settings for Specific Data Object:			Results in Generated Files:		
Storage Class Setting	**Data Def.**	**Data Dec.**	**Def. File**	**Owner**	**Header File**	**Where Data Def. Is**	**Where Data Dec. Is**	**Dec. Inclusion**
`Global`, `Const`, `Const-Volatile`, `Volatile`	`src`	D/C	null	null	`hdr.h`	`src.c`	`hdr.h`	Note 11
`Global`, `Const`, `Const-Volatile`, `Volatile`	`sep`	D/C	null	null	`hdr.h`	`gbl.c`	`hdr.h`	Note 11
`Global`, `Const`, `Const-Volatile`, `Volatile`	D/C	`auto`	null	External owner	null	External user-supplied file	`model.h`	Note 1
`Global`, `Const`, `Const-Volatile`, `Volatile`	D/C	`src`	null	External owner	null	External user-supplied file	`src.c`	Note 8
`Global`, `Const`, `Const-Volatile`, `Volatile`	D/C	`sep`	null	External owner	null	External user-supplied file	`gbl.h`	Note 9
`Global`, `Const`, `Const-Volatile`, `Volatile`	D/C	D/C	null	External owner	`header.h`	External user-supplied file	`hdr.h`	Note 11
`Global`, `Const`, `Const-Volatile`, `Volatile`	D/C	D/C	null	External owner	`header.h`	External user-supplied file	`hdr.h`	Note 11
mpt Custom Storage Class: Exported Data:								
`ExportTo-File`	`auto`	`auto`	null	null	null	`model.c`	`model.h`	Note 1

	Global Settings:		Override Settings for Specific Data Object:			Results in Generated Files:		
Storage Class Setting	**Data Def.**	**Data Dec.**	**Def. File**	**Owner**	**Header File**	**Where Data Def. Is**	**Where Data Dec. Is**	**Dec. Inclusion**
`ExportTo-File`	`src`	`auto`	null	null	null	`src.c`	`model.h`	Note 1
`ExportTo-File`	`sep`	`auto`	null	null	null	`gbl.c`	`model.h`	Note 1
`ExportTo-File`	`auto`	`src`	null	null	null	`model.c`	`src.c`	Note 8
`ExportTo-File`	`src`	`src`	null	null	null	`src.c`	`src.c`	Note 8
`ExportTo-File`	`sep`	`src`	null	null	null	`gbl.c`	`src.c`	Note 8
`ExportTo-File`	`auto`	`sep`	null	null	null	`model.c`	`gbl.h`	Note 9
`ExportTo-File`	`src`	`sep`	null	null	null	`src.c`	`gbl.h`	Note 9
`ExportTo-File`	`sep`	`sep`	null	null	null	`gbl.c`	`gbl.h`	Note 9
`ExportTo-File`	D/C	D/C	`data.c`	null	null	`data.c`	See Note 10.	Note 10
`ExportTo-File`	D/C	D/C	`data.c`	null	`hdr.h`	`model.c`	`hdr.h`	Note 11
`ExportTo-File`	`auto`	D/C	null	null	`hdr.h`	`src.c`	`hdr.h`	Note 11
`ExportTo-File`	`sep`	D/C	null	null	`hdr.h`	`gbl.c`	`hdr.h`	Note 11
Simulink Custom Storage Class: Default, Const, ConstVolatile, Volatile:								
`Default`, `Const`, `Const-Volatile`, `Volatile`	`auto`	`auto`	N/A	N/A	N/A	`model.c`	`model.h`	Note 1
`Default`, `Const`, `Const-Volatile`, `Volatile`	`src`	`auto`	N/A	N/A	N/A	`src.c`	`model.h`	Note 1
`Default`, `Const`, `Const-Volatile`, `Volatile`	`sep`	`auto`	N/A	N/A	N/A	`gbl.c`	`model.h`	Note 1

	Global Settings:		Override Settings for Specific Data Object:			Results in Generated Files:		
Storage Class Setting	**Data Def.**	**Data Dec.**	**Def. File**	**Owner**	**Header File**	**Where Data Def. Is**	**Where Data Dec. Is**	**Dec. Inclusion**
`Default`, `Const`, `Const-Volatile`, `Volatile`	`auto`	`src`	N/A	N/A	N/A	`model.c`	`src.c`	Note 8
`Default`, `Const`, `Const-Volatile`, `Volatile`	`src`	`src`	N/A	N/A	N/A	`src.c`	`src.c`	Note 8
`Default`, `Const`, `Const-Volatile`, `Volatile`	`sep`	`src`	N/A	N/A	N/A	`gbl.c`	`src.c`	Note 8
`Default`, `Const`, `Const-Volatile`, `Volatile`	`auto`	`sep`	N/A	N/A	N/A	`model.c`	`gbl.h`	Note 9
`Default`, `Const`, `Const-Volatile`, `Volatile`	`src`	`sep`	N/A	N/A	N/A	`src.c`	`gbl.h`	Note 9
`Default`, `Const`, `Const-Volatile`, `Volatile`	`sep`	`sep`	N/A	N/A	N/A	`gbl.c`	`gbl.h`	Note 9
Simulink Custom Storage Class: Exported Data:								
`ExportTo-File`	`auto`	`auto`	N/A	N/A	null	`model.c`	`model.h`	Note 1
`ExportTo-File`	`src`	`auto`	N/A	N/A	null	`src.c`	`model.h`	Note 1
`ExportTo-File`	`sep`	`auto`	N/A	N/A	null	`gbl.c`	`model.h`	Note 1
`ExportTo-File`	`auto`	`src`	N/A	N/A	null	`model.c`	`src.c`	Note 8

	Global Settings:		Override Settings for Specific Data Object:			Results in Generated Files:		
Storage Class Setting	**Data Def.**	**Data Dec.**	**Def. File**	**Owner**	**Header File**	**Where Data Def. Is**	**Where Data Dec. Is**	**Dec. Inclusion**
`ExportTo-File`	`src`	`src`	N/A	N/A	null	`src.c`	`src.c`	Note 8
`ExportTo-File`	`sep`	`src`	N/A	N/A	null	`gbl.c`	`src.c`	Note 8
`ExportTo-File`	`auto`	`sep`	N/A	N/A	null	`model.c`	`gbl.h`	Note 9
`ExportTo-File`	`src`	`sep`	N/A	N/A	null	`src.c`	`gbl.h`	Note 9
`ExportTo-File`	`sep`	`sep`	N/A	N/A	null	`gbl.c`	`gbl.h`	Note 9
`ExportTo-File`	`auto`	D/C	N/A	N/A	`hdr.h`	`model.c`	`hdr.h`	Note 11
`ExportTo-File`	`src`	D/C	N/A	N/A	`hdr.h`	`src.c`	`hdr.h`	Note 11
`ExportTo-File`	`sep`	D/C	N/A	N/A	`hdr.h`	`gbl.c`	`hdr.h`	Note 11

Notes

In the previous table:

- A Declaration Inclusion Approach is a file in which the header file that contains the data declarations is included.
- D/C stands for don't care.
- Dec stands for declaration.
- Def stands for definition.
- `gbl` stands for global.
- `hdr` stands for header.
- N/A stands for not applicable.
- null stands for field is blank.
- `sep` stands for separate.

Note 1: `model.h` is included directly in all source files.

Note 2: `model_private.h` is included directly in all source files.

Note 3: `extern` is included in `model_private.h`, which is in `source.c`.

Note 4: `header.h` is included in `model_private.h`, which is in `source.c`.

Note 5: `model.h` is included directly in all source files that use `#define`.

Note 6: `header.h` is included in `model.h`, which is in source files that use `#define`.

Note 7: `model.h` is included in all `source.c` files.

Note 8: `extern` is inlined in source files where data is used.

Note 9: `global.h` is included in `model.h`, which is in all source files.

Note 10: When you specify a definition filename for a data object, a header file is not generated for that data object. The code generator declares the data object according to the data placement priorities.

Note 11: `header.h` is included in `model.h`, which is in all source files.

Note 12: Signal: Either not defined because it is expression folded, or local data, or defined in a structure in `model.c`, all depending on model's code generation settings. Parameter: Either inlined in the code, or defined in `model_data.c`.

Note 13: Signal: In a structure that is defined in `model.c`. Parameter: In a structure that is defined in `model_data.c`.

Specify Default `#include` Syntax for Data Header Files

To control the file placement of a data item such as a signal line or block state in the generated code, you can apply a custom storage class to the data item (see "Introduction to Custom Storage Classes" on page 23-2). You then use the `HeaderFile` custom attribute to specify the generated or custom header file that contains the declaration of the data.

To reduce maintenance effort and data entry, when you specify `HeaderFile`, you can omit delimiters (`"` or `<>`) and use only the file name. You can then control the default delimiters that the generated code uses for the corresponding `#include` directives. To use angle brackets by default, set **Configuration Parameters** > **Code Generation** > **Code Placement** > **#include file delimiters** to `#include <header.h>`.

Related Examples

- "Introduction to Custom Storage Classes" on page 23-2

- "Block Parameter Representation in the Generated Code" on page 19-47
- "Signal Representation in Generated Code" on page 19-112

Enhance Readability of Code for Flow Charts

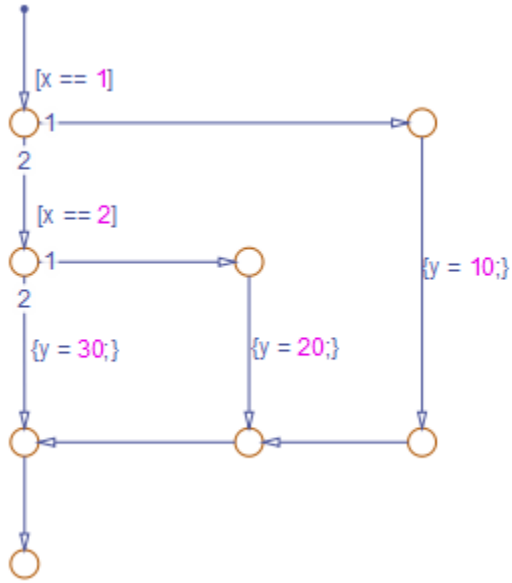
In this section...

Appearance of Generated Code for Flow Charts

If you have an Embedded Coder license and you generate code for models that include Stateflow objects, the code from a flow chart resembles the samples that follow.

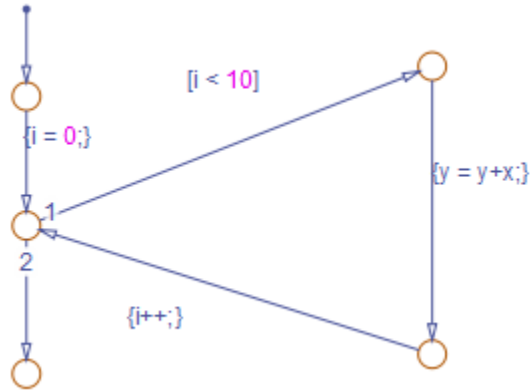
The following characteristics apply:

- By default, the generated code uses `if-elseif-else` statements to represent `switch` patterns. To convert the code to use `switch-case` statements, see “Convert If-Elseif-Else Code to Switch-Case Statements” on page 36-130.
- By default, variables that appear in the flow chart do not retain their names in the generated code. Modified identifiers make sure that no naming conflicts occur.
- Traceability comments for the transitions appear between each set of `/*` and `*/` markers. To learn more about traceability, see “Trace Stateflow Objects in Generated Code” on page 61-10.



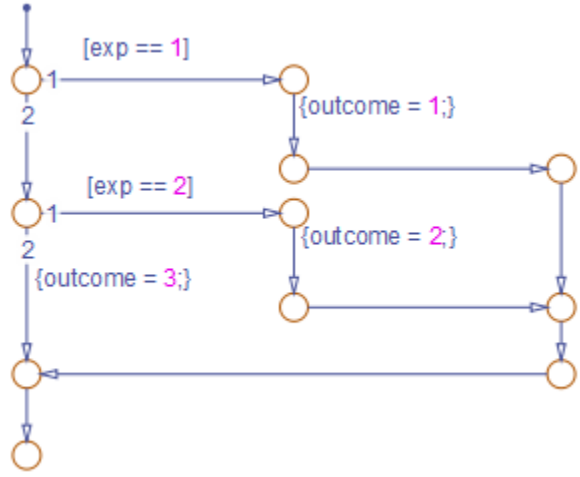

```
if (modelname_U.In1 == 1.0) {
  /* Transition: '<S1>:11' */
  /* Transition: '<S1>:12' */
  modelname_Y.Out1 = 10.0;

  /* Transition: '<S1>:15' */
  /* Transition: '<S1>:16' */
} else {
  /* Transition: '<S1>:10' */
  if (modelname_U.In1 == 2.0) {
    /* Transition: '<S1>:13' */
    /* Transition: '<S1>:14' */
    modelname_Y.Out1 = 20.0;

    /* Transition: '<S1>:16' */
  } else {
    /* Transition: '<S1>:17' */
    modelname_Y.Out1 = 30.0;
  }
}
```

Sample Code for a Decision Logic Pattern




```
for (sf_i = 0; sf_i < 10; sf_i++) {
  /* Transition: '<S1>:40' */
  /* Transition: '<S1>:41' */
  modelname_B.y = modelname_B.y +
    modelname_U.In1;

  /* Transition: '<S1>:39' */
}
```

Sample Code for an Iterative Loop Pattern



```
if (modelname_U.In1 == 1.0) {
  /* Transition: '<S1>:149' */
  /* Transition: '<S1>:150' */
  modelname_Y.Out1 = 1.0;

  /* Transition: '<S1>:151' */
  /* Transition: '<S1>:152' */
  /* Transition: '<S1>:158' */
  /* Transition: '<S1>:159' */
} else {
  /* Transition: '<S1>:156' */
  if (modelname_U.In1 == 2.0) {
    /* Transition: '<S1>:153' */
    /* Transition: '<S1>:154' */
    modelname_Y.Out1 = 2.0;

    /* Transition: '<S1>:155' */
    /* Transition: '<S1>:158' */
    /* Transition: '<S1>:159' */
  } else {
    /* Transition: '<S1>:161' */
    modelname_Y.Out1 = 3.0;
  }
}
```

Sample Code for a Switch Pattern

Convert If-Elseif-Else Code to Switch-Case Statements

When you generate code for embedded real-time targets, you can choose to convert `if-elseif-else` code to `switch-case` statements. This conversion can enhance readability of the code. For example, when a flow chart contains a long list of conditions, the `switch-case` structure:

- Reduces the use of parentheses and braces
- Minimizes repetition in the generated code

How to Convert If-Elseif-Else Code to Switch-Case Statements

The following procedure describes how to convert generated code for the flow chart from `if-elseif-else` to `switch-case` statements.

Step	Task	Reference
1	Verify that your flow chart follows the rules for conversion.	"Verify the Contents of the Flow Chart" on page 36-134
2	Enable the conversion.	"Enable the Conversion" on page 36-135
3	Generate code for your model.	"Generate Code for Your Model" on page 36-136
4	Troubleshoot the generated code. • If you see `switch-case` statements for your flow chart, you can stop. • If you see `if-elseif-else` statements for your flow chart, update the chart and repeat the previous step.	"Troubleshoot the Generated Code" on page 36-136

Rules of Conversion

For the conversion to occur, the following rules must hold. LHS and RHS refer to the left-hand side and right-hand side of a condition, respectively.

Construct	Rules to Follow
Flow chart	Must have two or more *unique* conditions, in addition to a default. For more information, see "How the Conversion Handles Duplicate Conditions" on page 36-132.
Each condition	Must test equality only.
	Must use the same variable or expression for the LHS. **Note:** You can reverse the LHS and RHS.
Each LHS	Must be a single variable or expression.
	Cannot be a constant.
	Must have an integer or enumerated data type.
	Cannot have any side effects on simulation.

Construct	Rules to Follow
	For example, the LHS can read from but not write to global variables.
Each RHS	Must be a constant.
	Must have an integer or enumerated data type.

How the Conversion Handles Duplicate Conditions

If a flow chart has duplicate conditions, the conversion preserves only the first condition. The code discards the other instances of duplicate conditions.

After removal of duplicates, two or more unique conditions must exist. If not, the conversion does not occur and the code contains all duplicate conditions.

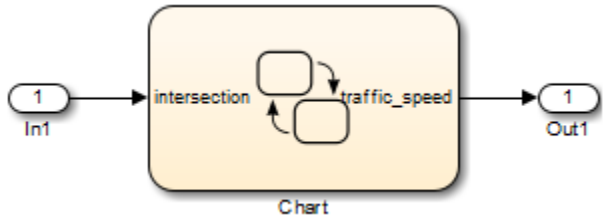
Example of Generated Code	Code After Conversion

`if (x == 1) {`
`  block1`
`} else if (x == 2) {`
`  block2`
`} else if (x == 1) {  // duplicate`
`  block3`
`} else if (x == 3) {`
`  block4`
`} else if (x == 1) {  // duplicate`
`  block5`
`} else {`
`  block6`
`}`
	`switch (x) {`
	`  case 1:`
	`  block1; break;`
	`  case 2:`
	`  block2; break;`
	`  case 3:`
	`  block4; break;`
	`  default:`
	`  block6; break;`
	`}`

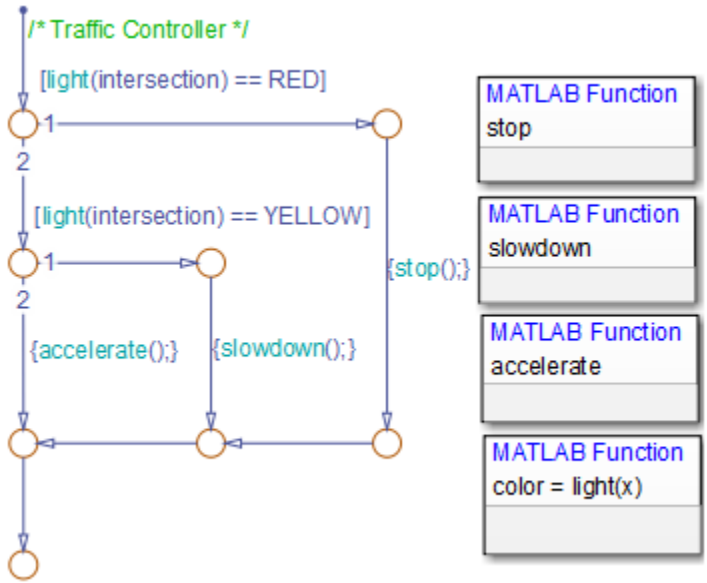
`if (x == 1) {`
`  block1`
`} else if (x == 1) {  // duplicate`
`  block2`
`} else {`
`  block3`
`}`
	No change, because only one unique condition exists

Example of Converting Code to Switch-Case Statements

Suppose that you have the following model with a single chart.



The chart contains a flow chart and four MATLAB functions:



The MATLAB functions in the chart contain the code in the following table. In each case, the **Function Inline Option** is `Auto`. For more information about function inlining, see "Specify Graphical Function Properties" (Stateflow).

MATLAB Function	Code

`stop`	`function stop`
	`%#codegen`
	`coder.extrinsic('disp');`
	`disp('Not moving.')`

MATLAB Function	Code
	`traffic_speed = 0;`

`slowdown`	`function slowdown`
	`%#codegen`
	`coder.extrinsic('disp')`
	`disp('Slowing down.')`
	
	`traffic_speed = 1;`

`accelerate`	`function accelerate`
	`%#codegen`
	`coder.extrinsic('disp');`
	`disp('Moving along.')`
	
	`traffic_speed = 2;`

`light`	`function color = light(x)`
	`%#codegen`
	`if (x < 20)`
	`    color = TrafficLights.GREEN;`
	`elseif (x >= 20 && x < 25)`
	`    color = TrafficLights.YELLOW;`
	`else`
	`    color = TrafficLights.RED;`
	`end`

The output `color` of the function `light` uses the enumerated type `TrafficLights`. The enumerated type definition in `TrafficLights.m` is:

```
classdef TrafficLights < Simulink.IntEnumType
  enumeration
    RED(0)
    YELLOW(5)
    GREEN(10)
  end
end
```

For more information, see "Define Enumerated Data in a Chart" (Stateflow).

Verify the Contents of the Flow Chart

Check that the flow chart in your chart follows the rules in "Rules of Conversion" on page 36-131.

Construct	How the Construct Follows the Rules
Flow chart	Two unique conditions exist, in addition to the default: • `[light(intersection) == RED]` • `[light(intersection) == YELLOW]`
Each condition	Each condition: • Tests equality • Uses the same function call `light(intersection)` for the LHS
Each LHS	Each LHS: • Contains a single expression • Is the output of a function call and therefore not a constant • Is of enumerated type `TrafficLights`, which you define in `TrafficLights.m` on the MATLAB path (see "Define Enumerated Data in a Chart" (Stateflow)) • Uses a function call that does not have side effects
Each RHS	Each RHS: • Is an enumerated value and therefore a constant • Is of enumerated type `TrafficLights`

Enable the Conversion

1. Open the Model Configuration Parameters dialog box.
2. In the **Code Generation** pane, select `ert.tlc` for the **System target file**.

   This step specifies an ERT-based target for your model.
3. In the **Code Generation** > **Code Style** pane, select the **Convert if-elseif-else patterns to switch-case statements** check box.

Tip: This conversion works on a per-model basis. If you select this check box, the conversion applies to:

- Flow charts in all charts of a model
- MATLAB functions in all charts of a model

- All MATLAB Function blocks in that model

Generate Code for Your Model

In the model, select **Code > C/C++ Code > Build Model** to generate source code from the model.

Troubleshoot the Generated Code

The generated code for the flow chart appears something like this:

```
if (sf_color == RED) {
  /* Transition: '<S1>:11' */
  /* Transition: '<S1>:12' */
  /* MATLAB Function 'stop': '<S1>:23' */
  /* '<S1>:23:6' */
  rtb_traffic_speed = 0;

  /* Transition: '<S1>:15' */
  /* Transition: '<S1>:16' */
} else {
  /* Transition: '<S1>:10' */
  /* MATLAB Function 'light': '<S1>:19' */
  if (ifelse_using_enums_U.In1 < 20.0) {
    /* '<S1>:19:3' */
    /* '<S1>:19:4' */
    sf_color = GREEN;
  } else if ((ifelse_using_enums_U.In1 >= 20.0) &&
             (ifelse_using_enums_U.In1 < 25.0)) {
    /* '<S1>:19:5' */
    /* '<S1>:19:6' */
    sf_color = YELLOW;
  } else {
    /* '<S1>:19:8' */
    sf_color = RED;
  }

  if (sf_color == YELLOW) {
    /* Transition: '<S1>:13' */
    /* Transition: '<S1>:14' */
    /* MATLAB Function 'slowdown': '<S1>:24' */
    /* '<S1>:24:6' */
    rtb_traffic_speed = 1;
```

```
    /* Transition: '<S1>:16' */
  } else {
    /* Transition: '<S1>:17' */
    /* MATLAB Function 'accelerate': '<S1>:25' */
    /* '<S1>:25:6' */
    rtb_traffic_speed = 2;
  }
}
```

Because the MATLAB function `light` appears inlined, inequality comparisons appear in these lines of code:

```
if (ifelse_using_enums_U.In1 < 20.0) {
....
} else if ((ifelse_using_enums_U.In1 >= 20.0) &&
             (ifelse_using_enums_U.In1 < 25.0)) {
....
```

Because inequalities appear in the body of the `if-elseif-else` code for the flow chart, the conversion to `switch-case` statements does not occur. To prevent this behavior, do one of the following:

- Specify that the function `light` does not appear inlined. See "Change the Inlining Property for the Function" on page 36-137.
- Modify the flow chart. See "Modify the Flow Chart to Ensure Switch-Case Statements" on page 36-139.

Change the Inlining Property for the Function

If you do not want to modify your flow chart, change the inlining property for the function `light`:

1. Right-click the function box for `light` and select **Properties**.

   The properties dialog box appears.
2. For **Function Inline Option**, select `Function`.
3. Click **OK** to close the dialog box.

Note: You do not have to change the inlining property for the other three MATLAB functions in the chart. Because the flow chart does not call those functions during evaluation of conditions, the inlining property for those functions can remain `Auto`.

When you regenerate code for your model, the code for the flow chart now appears something like this:

```
switch (ifelse_using_enums_light(ifelse_using_enums_U.In1)) {
 case RED:
  /* Transition: '<S1>:11' */
  /* Transition: '<S1>:12' */
  /* MATLAB Function 'stop': '<S1>:23' */
  /* '<S1>:23:6' */
  ifelse_using_enums_Y.Out1 = 0.0;

  /* Transition: '<S1>:15' */
  /* Transition: '<S1>:16' */
  break;

 case YELLOW:
  /* Transition: '<S1>:10' */
  /* Transition: '<S1>:13' */
  /* Transition: '<S1>:14' */
  /* MATLAB Function 'slowdown': '<S1>:24' */
  /* '<S1>:24:6' */
  ifelse_using_enums_Y.Out1 = 1.0;

  /* Transition: '<S1>:16' */
  break;

 default:
  /* Transition: '<S1>:17' */
  /* MATLAB Function 'accelerate': '<S1>:25' */
  /* '<S1>:25:6' */
  ifelse_using_enums_Y.Out1 = 2.0;
  break;
}
```

Because the MATLAB function `light` no longer appears inlined, the conversion to `switch-case` statements occurs. The `switch-case` statements provide the following benefits to enhance readability:

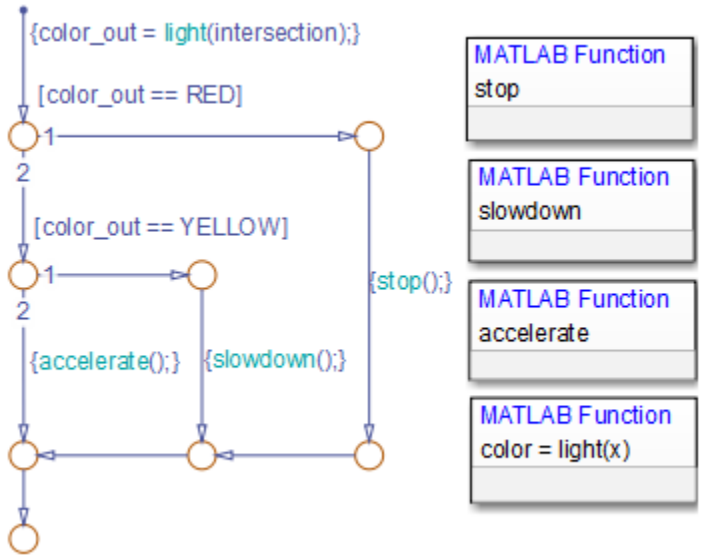
- The code reduces the use of parentheses and braces.
- The LHS expression `ifelse_using_enums_light(ifelse_using_enums_U.In1)` appears only once, minimizing repetition in the code.

Modify the Flow Chart to Ensure Switch-Case Statements

If you do not want to change the inlining property for the function `light`, modify your flow chart:

1. Add chart local data `color_out` with the enumerated type `TrafficLights`.
2. Replace each instance of `light(intersection)` with `color_out`.
3. Add the action `{color_out = light(intersection)}` to the default transition of the flow chart.

The chart should now look something like this:



When you regenerate code for your model, the code for the flow chart uses `switch-case` statements.

Generate Inlined Subsystem Code

You can configure a nonvirtual subsystem to inline the subsystem code with the model code. In the Subsystem Parameters dialog box, the **Function packaging** parameter specifies the format of the subsystem's generated code. This parameter has four settings: `Auto`, `Inline`, `Nonreusable function`, and `Reusable function`. The code generator can generate inlined code for the `Auto` or `Inline` settings.

The `Inline` setting explicitly directs the code generator to inline the subsystem code unconditionally.

The default `Auto` setting directs the code generator to generate the most efficient code for the subsystem based on the type and number of instances of the subsystem that exist in the model. When there is only one instance of a subsystem, the `Auto` setting inlines the subsystem code. When there are multiple instances of a subsystem, that is not too complex, the `Auto` setting inlines the code for each subsystem. Otherwise, the `Auto` setting generates a single copy of the function (as a reusable function). For a function-call subsystem with multiple callers, the `Auto` setting generates subsystem code that is consistent with the `Nonreusable function` setting.

Configure Subsystem to Inline Code

To configure your subsystem for inlining:

1. Right-click the Subsystem block. From the context menu, select **Block Parameters (Subsystem)**.
2. In the Subsystem Parameters dialog box, if the subsystem is virtual, select **Treat as atomic unit**. This option makes the subsystem nonvirtual. On the **Code Generation** tab, the **Function packaging** option is now available.
3. Click the **Code Generation** tab and select `Auto` or `Inline` from the **Function packaging** parameter.
4. Click **Apply** and close the dialog box.

   The border of the subsystem thickens, indicating that it is nonvirtual.

When you generate code from your model, the code generator inlines subsystem code within *`model`*`.c` or *`model`*`.cpp` (or in its parent system's source file). You can identify this code by system/block identification tags, such as:

```
/* Atomic SubSystem Block: <Root>/AtomicSubsys1 */
```

Exceptions to Inlining

There are certain cases in which the code generator does not inline a nonvirtual subsystem, even though you select the `Inline` setting.

- If a noninlined S-function calls a function-call subsystem, the code generator ignores the `Inline` setting. Because noninlined S-functions use function pointers to make function calls, the code generator must generate a function with all arguments present.
- In a feedback loop involving function-call subsystems, the code generator generates a function instead of inlined code for one of the subsystems. Based on the internal, sorted order of the subsystems, the code generator selects which subsystem to generate a function.
- If an S-function, an Async Interrupt, or a Task Sync block with the option `SS_OPTION_FORCE_NONINLINED_FCNCALL` set to `TRUE` calls a subsystem, the code generator generates a function instead of inlined code for the subsystem. The VxWorks block library (`vxlib1`), contains the user-defined Async Interrupt and Task Sync blocks.[7]

See Also

- "Code Generation of Subsystems" (Simulink Coder)
- "Generate Subsystem Code as Separate Function and Files" (Simulink Coder)
- "Generate Reusable Function for Identical Subsystems Within a Model" (Simulink Coder)

7. VxWorks is a registered trademark of Wind River Systems, Inc.

Code Replacement in Simulink Coder

What Is Code Replacement?

Code replacement is a technique to change the code that the code generator produces for functions and operators to meet application code requirements. For example, you can replace generated code to meet requirements such as:

- Optimization for a specific run-time environment, including, but not limited to, specific target hardware.
- Integration with existing application code.
- Compliance with a standard, such as AUTOSAR.
- Modification of code behavior, such as enabling or disabling nonfinite or inline support.
- Application- or project-specific code requirements, such as:
  - Elimination of `math.h`.
  - Elimination of system header files.
  - Elimination of calls to `memcpy` or `memset`.
  - Use of BLAS.
  - Use of a specific BLAS.

To apply this technique, configure the code generator to apply a code replacement library (CRL) during code generation. By default, the code generator does not apply a code replacement library. You can choose from the following libraries that MathWorks provides:

- GNU C99 extensions—GNU[8] gcc math library, which provides C99 extensions as defined by compiler option `-std=gnu99`.
- AUTOSAR 4.0—Produces code that more closely aligns with the AUTOSAR standard. Requires an Embedded Coder license.
- Intel IPP for x86-64 (Windows)—Generates calls to the Intel® Performance Primitives (IPP) library for the x86-64 Windows platform.
- Intel IPP/SSE for x86-64 (Windows)—Generates calls to the IPP and Streaming SIMD Extensions (SSE) libraries for the x86-64 Windows platform.
- Intel IPP for x86-64 (Windows using MinGW compiler)—Generates calls to the IPP library for the x86-64 Windows platform and MinGW compiler.

8. GNU is a registered trademark of the Free Software Foundation.

- Intel IPP/SSE for x86-64 (Windows using MinGW compiler)—Generates calls to the IPP and SSE libraries for the x86-64 Windows platform and MinGW compiler.
- Intel IPP for x86/Pentium (Windows)—Generates calls to the IPP library for the x86/Pentium Windows platform.
- Intel IPP/SSE for x86/Pentium (Windows)—Generates calls to the Intel Performance IPP and SSE libraries for the x86/Pentium Windows platform.
- Intel IPP for x86-64 (Linux)—Generates calls to the IPP library for the x86-64 Linux platform.
- Intel IPP/SSE with GNU99 extensions for x86-64 (Linux)—Generates calls to the GNU libraries for IPP and SSE, with GNU C99 extensions, for the x86-64 Linux platform.

Libraries that include GNU99 extensions are intended for use with the GCC compiler. If you use one of those libraries with another compiler, generated code might not compile.

Depending on the product licenses that you have, other libraries might be available . If you have an Embedded Coder license, you can view and choose from other libraries and you can create custom code replacement libraries.

Code Replacement Libraries

A *code replacement library* consists of one or more code replacement tables that specify application-specific implementations of functions and operators. For example, a library for a specific embedded processor specifies function and operator replacements that optimize generated code for that processor.

A *code replacement table* contains one or more *code replacement entries*, with each entry representing a potential replacement for a function or operator. Each entry maps a *conceptual representation* of a function or operator to an *implementation representation* and priority.

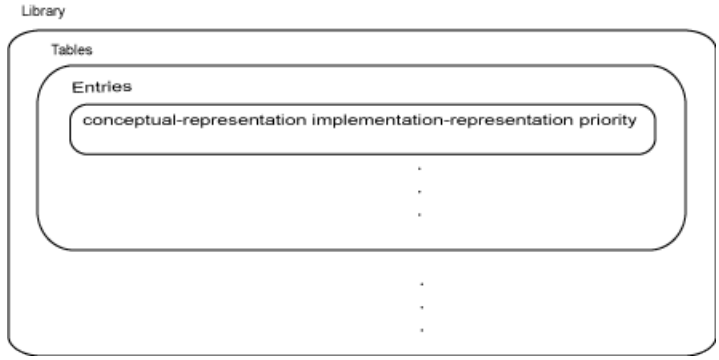


Table Entry Component	Description
Conceptual representation	Identifies the table entry and contains match criteria for the code generator. Consists of: • Function name or a key. The function name identifies most functions. For operators and some functions, a series of characters, called a key identifies a function or operator. For example, function name `'cos'` and operator key `'RTW_OP_ADD'`. • Conceptual arguments that observe code generator naming (`'y1'`, `'u1'`, `'u2'`, ...), with corresponding I/O types (output or input) and data types. • Other attributes, such as an algorithm, fixed-point saturation, and rounding modes, which identify matching criteria for the function or operator.
Implementation representation	Specifies replacement code. Consists of: • Function name. For example, `'cos_dbl'` or `'u8_add_u8_u8'`. • Implementation arguments, with corresponding I/O types (output or input) and data types. • Parameters that provide additional implementation details, such as header and source file names and paths of build resources.

Table Entry Component	Description
Priority	Defines the entry priority relative to other entries in the table. The value can range from 0 to 100, with 0 being the highest priority. If multiple entries have the same priority, the code generator uses the first match with that priority.

When the code generator looks for a match in a code replacement library, it creates and populates a *call site object* with the function or operator conceptual representation. If a match exists, the code generator uses the matched code replacement entry populated with the implementation representation and uses it to generate code.

The code generator searches the tables in a code replacement library for a match in the order that the tables appear in the library. If the code generator finds multiple matches within a table, the priority determines the match. The code generator uses a higher-priority entry over a similar entry with a lower priority.

Code Replacement Terminology

Term	Definition
Cache hit	A code replacement entry for a function or operator, defined in the specified code replacement library, for which the code generator finds a match.
Cache miss	A conceptual representation of a function or operator for which the code generator does not find a match.
Call site object	Conceptual representation of a function or operator that the code generator uses when it encounters a call site for a function or operator. The code generator uses the object to query the code replacement library for a conceptual representation match. If a match exists, the code generator returns a code replacement object, fully populated with the conceptual representation, implementation representation, and priority, and uses that object to generate replacement code.
Code replacement library	One or more code replacement tables that specify application-specific implementations of functions

Term	Definition
	and operators. When configured to use a code replacement library, the code generator uses criteria defined in the library to search for matches. If a match is found, the code generator replaces code that it generates by default with application-specific code defined in the library.
Code replacement table	One or more code replacement table entries. Provides a way to group related or shared entries for use in different libraries.
Code replacement entry	Represents a potential replacement for a function or operator. Maps a conceptual representation of a function or operator to an implementation representation and priority.
Conceptual argument	Represents an input or output argument for a function or operator being replaced. Conceptual arguments observe naming conventions (`'y1'`, `'u1'`, `'u2'`, ...) and data types familiar to the code generator.
Conceptual representation	Represents match criteria that the code generator uses to qualify functions and operators for replacement. Consists of: • Function or operator name or key • Conceptual arguments with type, dimension, and complexity specification for inputs and output • Attributes, such as an algorithm and fixed-point saturation and rounding modes
Implementation argument	Represents an input or output argument for a C or C++ replacement function. Implementation arguments observe C/C++ name and data type specifications.

Term	Definition
Implementation representation	Specifies C or C++ replacement function prototype. Consists of: • Function name (for example, `'cos_dbl'` or `'u8_add_u8_u8'`) • Implementation arguments specifying type, type qualifiers, and complexity for the function inputs and output • Parameters that provide build information, such as header and source file names and paths of build resources and compile and link flags
Key	Identifies a function or operator that is being replaced. A function name or key appears in the conceptual representation of a code replacement entry. The key `RTW_OP_ADD` identifies the addition operator.
Priority	Defines the match priority for a code replacement entry relative to other entries, which have the same name and conceptual argument list, within a code replacement library. The priority can range from 0 to 100, with 0 being the highest priority. The default is 100. If a library provides two implementations for a function or operator, the implementation with the higher priority shadows the one with the lower priority.

Code Replacement Limitations

Code replacement verification — It is possible that code replacement behaves differently than you expect. For example, data types that you observe in code generator input might not match what the code generator uses as intermediate data types during an operation. Verify code replacements by examining generated code.

Code replacement for matrices — Code replacement libraries do not support Dynamic and Symbolic sized matrices.

Related Examples

- “Choose a Code Replacement Library” (Simulink Coder)
- “Replace Code Generated from Simulink Models” (Simulink Coder)

Choose a Code Replacement Library

In this section...

About Choosing a Code Replacement Library

By default, the code generator does not use a code replacement library.

If you are considering using a code replacement library:

1. Explore available libraries. Identify one that best meets your application needs.
   - Consider the lists of application code replacement requirements and libraries that MathWorks provides in "What Is Code Replacement?" on page 38-2.
   - See "Explore Available Code Replacement Libraries" on page 37-9.
2. Explore the contents of the library. See "Explore Code Replacement Library Contents" on page 37-9.

If you do not find a suitable library and you have an Embedded Coder license, you can create a custom code replacement library.

Explore Available Code Replacement Libraries

Select the "Code replacement library" (Simulink Coder) to use for code generation from the **Configuration Parameters > Code Generation > Interface** pane (Simulink Coder). To view a description of a library, select and hover your cursor over the library name. A tooltip describes the library and lists the tables that it contains. The tooltip lists the tables in the order that the code generator searches for a function or operator match.

Explore Code Replacement Library Contents

Use the Code Replacement Viewer to explore the content of a code replacement library.

1. At the command prompt, type `crviewer`.

```
>> crviewer
```

The viewer opens. To view the content of a specific library, specify the name of the library as an argument in single quotes. For example:

```
>> crviewer('GNU C99 extensions')
```

2. In the left pane, select the name of a library. The viewer displays information about the library in the right pane.
3. In the left pane, expand the library, explore the list of tables it contains, and select a table from the list. In the middle pane, the viewer displays the function and operator entries that are in that table, along with abbreviated information for each entry.
4. In the middle pane, select a function or operator. The viewer displays information from the table entry in the right pane.

   If you select an operator entry that specifies net slope fixed-point parameters (instantiated from entry class `RTW.TflCOperationEntryGenerator` or `RTW.TflCOperationEntryGenerator_NetSlope`), the viewer displays an additional tab that shows fixed-point settings.

See Code Replacement Viewer for details on what the viewer displays.

Related Examples

- "What Is Code Replacement?" (Simulink Coder)
- "Replace Code Generated from Simulink Models" (Simulink Coder)

Replace Code Generated from Simulink Models

This example shows how to replace generated code, using a code replacement library. Code replacement is a technique you can use to change the code that the code generator produces for functions and operators to meet application code requirements.

Prepare for Code Replacement

1. Make sure that MATLAB, Simulink, Simulink Coder, and a C compiler are installed on your system. Some code replacement libraries available in your development environment can also require Embedded Coder.

   To install MathWorks products, see the MATLAB installation documentation. If you have installed MATLAB and want to see which other MathWorks products are installed, in the Command Window, enter `ver` .
2. Identify an existing or create a Simulink model for which you want the code generator to replace code.

Choose a Code Replacement Library

If you are not sure which library to use, explore the available libraries.

Configure Code Generator To Use Code Replacement Library

1. Configure the code generator to apply a code replacement library during code generation for the model. Do one of the following:

   - In the Configuration Parameters dialog box, on the **Code Generation** > **Interface** pane, select a library for the “Code replacement library” (Simulink Coder) parameter.
   - Set the `CodeReplacementLibrary` parameter at the command line or programmatically.
2. Configure the code generator to produce code only (not build an executable) so you can verify your code replacements before building an executable. Do one of the following:

   - In the Configuration Parameters dialog box, on the **Code Generation** pane, select “Generate code only” (Simulink Coder).
   - Set the `GenCodeOnly` parameter at the command line or programmatically.

Include Code Replacement Information In Code Generation Report

If you have an Embedded Coder license, you can configure the code generator to include a code replacement section in the code generation report. The additional information can help you verify code replacements.

1. Configure the code generator to generate a report. In the Configuration Parameters dialog box, on the **Code Generation > Report** pane, select "Create code generation report" (Simulink Coder). Consider having the report open automatically. Select "Open report automatically" (Simulink Coder).
2. Include the code replacement section in the report. On the **All Parameters** tab, select "Summarize which blocks triggered code replacements" (Simulink Coder).

Generate Replacement Code

Generate C/C++ code from the model and, if you configured the code generator accordingly, a code generation report. For example, in the model window, press **Ctrl+B**.

The code generator produces the code and displays the report.

Verify Code Replacements

Verify code replacements by examining the generated code. It is possible that code replacement behaves differently than you expect. For example, data types that you observe in the code generator input might not match what the code generator uses as intermediate data types during an operation.

Related Examples

- "What Is Code Replacement?" (Simulink Coder)
- "Choose a Code Replacement Library" (Simulink Coder)
- "Code Generation Configuration" (Simulink Coder)

Code Replacement for Simulink Models in Embedded Coder

What Is Code Replacement?

Code replacement is a technique to change the code that the code generator produces for functions and operators to meet application code requirements. For example, you can replace generated code to meet requirements such as:

- Optimization for a specific run-time environment, including, but not limited to, specific target hardware.
- Integration with existing application code.
- Compliance with a standard, such as AUTOSAR.
- Modification of code behavior, such as enabling or disabling nonfinite or inline support.
- Application- or project-specific code requirements, such as:
  - Elimination of `math.h`.
  - Elimination of system header files.
  - Elimination of calls to `memcpy` or `memset`.
  - Use of BLAS.
  - Use of a specific BLAS.

To apply this technique, configure the code generator to apply a code replacement library (CRL) during code generation. By default, the code generator does not apply a code replacement library. You can choose from the following libraries that MathWorks provides:

- GNU C99 extensions—GNU[9] gcc math library, which provides C99 extensions as defined by compiler option `-std=gnu99`.
- AUTOSAR 4.0—Produces code that more closely aligns with the AUTOSAR standard. Requires an Embedded Coder license.
- Intel IPP for x86-64 (Windows)—Generates calls to the Intel Performance Primitives (IPP) library for the x86-64 Windows platform.
- Intel IPP/SSE for x86-64 (Windows)—Generates calls to the IPP and Streaming SIMD Extensions (SSE) libraries for the x86-64 Windows platform.
- Intel IPP for x86-64 (Windows using MinGW compiler)—Generates calls to the IPP library for the x86-64 Windows platform and MinGW compiler.

9. GNU is a registered trademark of the Free Software Foundation.

- Intel IPP/SSE for x86-64 (Windows using MinGW compiler)—Generates calls to the IPP and SSE libraries for the x86-64 Windows platform and MinGW compiler.
- Intel IPP for x86/Pentium (Windows)—Generates calls to the IPP library for the x86/Pentium Windows platform.
- Intel IPP/SSE for x86/Pentium (Windows)—Generates calls to the Intel Performance IPP and SSE libraries for the x86/Pentium Windows platform.
- Intel IPP for x86-64 (Linux)—Generates calls to the IPP library for the x86-64 Linux platform.
- Intel IPP/SSE with GNU99 extensions for x86-64 (Linux)—Generates calls to the GNU libraries for IPP and SSE, with GNU C99 extensions, for the x86-64 Linux platform.

Libraries that include GNU99 extensions are intended for use with the GCC compiler. If you use one of those libraries with another compiler, generated code might not compile.

Depending on the product licenses that you have, other libraries might be available . If you have an Embedded Coder license, you can view and choose from other libraries and you can create custom code replacement libraries.

Code Replacement Libraries

A *code replacement library* consists of one or more code replacement tables that specify application-specific implementations of functions and operators. For example, a library for a specific embedded processor specifies function and operator replacements that optimize generated code for that processor.

A *code replacement table* contains one or more *code replacement entries*, with each entry representing a potential replacement for a function or operator. Each entry maps a *conceptual representation* of a function or operator to an *implementation representation* and priority.

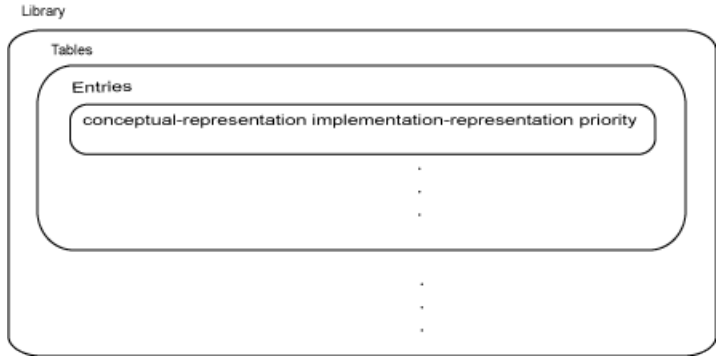


Table Entry Component	Description
Conceptual representation	Identifies the table entry and contains match criteria for the code generator. Consists of: • Function name or a key. The function name identifies most functions. For operators and some functions, a series of characters, called a key identifies a function or operator. For example, function name `'cos'` and operator key `'RTW_OP_ADD'`. • Conceptual arguments that observe code generator naming (`'y1'`, `'u1'`, `'u2'`, ...), with corresponding I/O types (output or input) and data types. • Other attributes, such as an algorithm, fixed-point saturation, and rounding modes, which identify matching criteria for the function or operator.
Implementation representation	Specifies replacement code. Consists of: • Function name. For example, `'cos_dbl'` or `'u8_add_u8_u8'`. • Implementation arguments, with corresponding I/O types (output or input) and data types. • Parameters that provide additional implementation details, such as header and source file names and paths of build resources.

Table Entry Component	Description
Priority	Defines the entry priority relative to other entries in the table. The value can range from 0 to 100, with 0 being the highest priority. If multiple entries have the same priority, the code generator uses the first match with that priority.

When the code generator looks for a match in a code replacement library, it creates and populates a *call site object* with the function or operator conceptual representation. If a match exists, the code generator uses the matched code replacement entry populated with the implementation representation and uses it to generate code.

The code generator searches the tables in a code replacement library for a match in the order that the tables appear in the library. If the code generator finds multiple matches within a table, the priority determines the match. The code generator uses a higher-priority entry over a similar entry with a lower priority.

Code Replacement Terminology

Term	Definition
Cache hit	A code replacement entry for a function or operator, defined in the specified code replacement library, for which the code generator finds a match.
Cache miss	A conceptual representation of a function or operator for which the code generator does not find a match.
Call site object	Conceptual representation of a function or operator that the code generator uses when it encounters a call site for a function or operator. The code generator uses the object to query the code replacement library for a conceptual representation match. If a match exists, the code generator returns a code replacement object, fully populated with the conceptual representation, implementation representation, and priority, and uses that object to generate replacement code.
Code replacement library	One or more code replacement tables that specify application-specific implementations of functions

Term	Definition
	and operators. When configured to use a code replacement library, the code generator uses criteria defined in the library to search for matches. If a match is found, the code generator replaces code that it generates by default with application-specific code defined in the library.
Code replacement table	One or more code replacement table entries. Provides a way to group related or shared entries for use in different libraries.
Code replacement entry	Represents a potential replacement for a function or operator. Maps a conceptual representation of a function or operator to an implementation representation and priority.
Conceptual argument	Represents an input or output argument for a function or operator being replaced. Conceptual arguments observe naming conventions (`'y1'`, `'u1'`, `'u2'`, ...) and data types familiar to the code generator.
Conceptual representation	Represents match criteria that the code generator uses to qualify functions and operators for replacement. Consists of: • Function or operator name or key • Conceptual arguments with type, dimension, and complexity specification for inputs and output • Attributes, such as an algorithm and fixed-point saturation and rounding modes
Implementation argument	Represents an input or output argument for a C or C++ replacement function. Implementation arguments observe C/C++ name and data type specifications.

Term	Definition
Implementation representation	Specifies C or C++ replacement function prototype. Consists of: • Function name (for example, `'cos_dbl'` or `'u8_add_u8_u8'`) • Implementation arguments specifying type, type qualifiers, and complexity for the function inputs and output • Parameters that provide build information, such as header and source file names and paths of build resources and compile and link flags
Key	Identifies a function or operator that is being replaced. A function name or key appears in the conceptual representation of a code replacement entry. The key `RTW_OP_ADD` identifies the addition operator.
Priority	Defines the match priority for a code replacement entry relative to other entries, which have the same name and conceptual argument list, within a code replacement library. The priority can range from 0 to 100, with 0 being the highest priority. The default is 100. If a library provides two implementations for a function or operator, the implementation with the higher priority shadows the one with the lower priority.

Code Replacement Limitations

Code replacement verification — It is possible that code replacement behaves differently than you expect. For example, data types that you observe in code generator input might not match what the code generator uses as intermediate data types during an operation. Verify code replacements by examining generated code.

Code replacement for matrices — Code replacement libraries do not support Dynamic and Symbolic sized matrices.

Related Examples

- “Choose a Code Replacement Library” on page 38-9
- “Replace Code Generated from Simulink Models” on page 38-11

Choose a Code Replacement Library

In this section...

About Choosing a Code Replacement Library

By default, the code generator does not use a code replacement library.

If you are considering using a code replacement library:

1. Explore available libraries. Identify one that best meets your application needs.
   - Consider the lists of application code replacement requirements and libraries that MathWorks provides in "What Is Code Replacement?" on page 38-2.
   - See "Explore Available Code Replacement Libraries" on page 37-9.
2. Explore the contents of the library. See "Explore Code Replacement Library Contents" on page 37-9.

If you do not find a suitable library and you have an Embedded Coder license, you can create a custom code replacement library.

Explore Available Code Replacement Libraries

Select the "Code replacement library" (Simulink Coder) to use for code generation from the **Configuration Parameters > Code Generation > Interface** pane (Simulink Coder). To view a description of a library, select and hover your cursor over the library name. A tooltip describes the library and lists the tables that it contains. The tooltip lists the tables in the order that the code generator searches for a function or operator match.

Explore Code Replacement Library Contents

Use the Code Replacement Viewer to explore the content of a code replacement library.

1. At the command prompt, type `crviewer`.

```
>> crviewer
```

The viewer opens. To view the content of a specific library, specify the name of the library as an argument in single quotes. For example:

```
>> crviewer('GNU C99 extensions')
```

2. In the left pane, select the name of a library. The viewer displays information about the library in the right pane.
3. In the left pane, expand the library, explore the list of tables it contains, and select a table from the list. In the middle pane, the viewer displays the function and operator entries that are in that table, along with abbreviated information for each entry.
4. In the middle pane, select a function or operator. The viewer displays information from the table entry in the right pane.

   If you select an operator entry that specifies net slope fixed-point parameters (instantiated from entry class `RTW.TflCOperationEntryGenerator` or `RTW.TflCOperationEntryGenerator_NetSlope`), the viewer displays an additional tab that shows fixed-point settings.

See Code Replacement Viewer for details on what the viewer displays.

Related Examples

- “What Is Code Replacement?” on page 38-2
- “Replace Code Generated from Simulink Models” on page 38-11

Replace Code Generated from Simulink Models

This example shows how to replace generated code, using a code replacement library. Code replacement is a technique you can use to change the code that the code generator produces for functions and operators to meet application code requirements.

Prepare for Code Replacement

1. Make sure that MATLAB, Simulink, Simulink Coder, and a C compiler are installed on your system. Some code replacement libraries available in your development environment can also require Embedded Coder.

   To install MathWorks products, see the MATLAB installation documentation. If you have installed MATLAB and want to see which other MathWorks products are installed, in the Command Window, enter `ver` .
2. Identify an existing or create a Simulink model for which you want the code generator to replace code.

Choose a Code Replacement Library

If you are not sure which library to use, explore the available libraries.

Configure Code Generator To Use Code Replacement Library

1. Configure the code generator to apply a code replacement library during code generation for the model. Do one of the following:

   - In the Configuration Parameters dialog box, on the **Code Generation** > **Interface** pane, select a library for the “Code replacement library” (Simulink Coder) parameter.
   - Set the `CodeReplacementLibrary` parameter at the command line or programmatically.
2. Configure the code generator to produce code only (not build an executable) so you can verify your code replacements before building an executable. Do one of the following:

   - In the Configuration Parameters dialog box, on the **Code Generation** pane, select “Generate code only” (Simulink Coder).
   - Set the `GenCodeOnly` parameter at the command line or programmatically.

Include Code Replacement Information In Code Generation Report

If you have an Embedded Coder license, you can configure the code generator to include a code replacement section in the code generation report. The additional information can help you verify code replacements.

1. Configure the code generator to generate a report. In the Configuration Parameters dialog box, on the **Code Generation > Report** pane, select "Create code generation report" (Simulink Coder). Consider having the report open automatically. Select "Open report automatically" (Simulink Coder).
2. Include the code replacement section in the report. On the **All Parameters** tab, select "Summarize which blocks triggered code replacements" (Simulink Coder).

Generate Replacement Code

Generate C/C++ code from the model and, if you configured the code generator accordingly, a code generation report. For example, in the model window, press **Ctrl+B**.

The code generator produces the code and displays the report.

Verify Code Replacements

Verify code replacements by examining the generated code. It is possible that code replacement behaves differently than you expect. For example, data types that you observe in the code generator input might not match what the code generator uses as intermediate data types during an operation.

Related Examples

- "What Is Code Replacement?" on page 38-2
- "Choose a Code Replacement Library" on page 38-9
- "Code Generation Configuration" (Simulink Coder)

Deployment

External Code Integration in Simulink Coder

The code generator includes multiple approaches for integrating legacy or custom code with generated code. *Legacy code* is existing handwritten code or code for environments that you integrate with code produced by the code generator. *Custom code* is legacy code or other user-specified lines of code that you include in the code generator build process. Collectively, legacy and custom code are called *external code*.

You integrate external code by importing existing external code into code produced by the code generator, exporting generated code into an existing external code base, or you can do both. For example, you can import code by calling an external function, by using the Legacy Code Tool, or place external code at specific locations in generated code by including Custom Code blocks in a model. When you import external code, the resulting generated code interfaces with generated scheduling code.

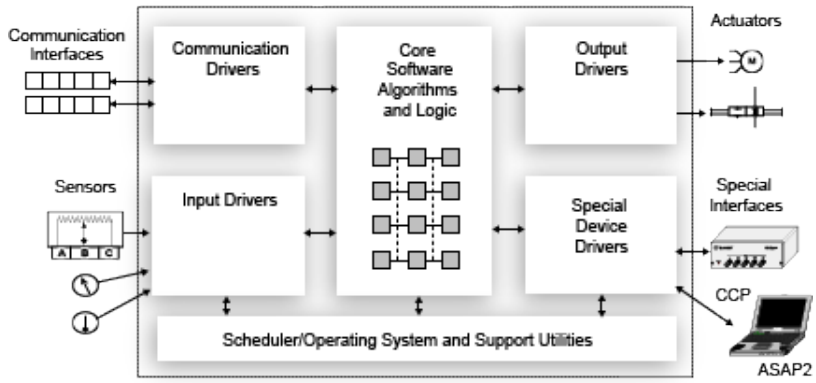
You can export generated code as a plug-in function for use in an external development environment. When you export generated code, you intend to interface that code manually with a scheduling mechanism in your application run-time environment.

For guidance on choosing an approach based on your application, see “Choose an External Code Integration Workflow” (Simulink Coder) .

What Is External Code Integration?

Software projects typically involve combining code from multiple sources. A typical system structure for a code generation application consists of a framework that combines code from multiple sources, including external code and code generated from Simulink models.

This figure shows an application that requires integration of existing driver code for hardware devices. The core software algorithms and logic can be a combination of code modules for external reusable algorithms integrated into the Simulink environment and code generated as part of an overall model design.



Several workflows and tools are available for you to integrate external and generated code. Each workflow identifies tooling for generating code that aligns interfaces, code appearance, and other factors, such as optimization between external and generated code.

More About

- “Choose an External Code Integration Workflow” on page 39-4

Choose an External Code Integration Workflow

In this section...

Completing these tasks helps you choose external code integration workflows and tooling that align with your project.

Task	Action	More Information
1	Partition your application, map algorithms to components, and identify integration points.	"Design Models for Generated Embedded Code Deployment" on page 1-2
2	Determine whether you can rely on scheduling code that the code generator produces, or whether you must integrate generated code with scheduling mechanisms that are specific to your run-time environment.	"Choose a Software Execution Framework" on page 39-4
3	Evaluate the characteristics of the external code that you are importing or to which you are exporting generated code.	"Evaluate Characteristics of External Code" on page 39-7
4	Identify integration requirements, which assists with choosing optimal tooling for your integration.	"Identify Integration Requirements" on page 39-8
5	Based on the results of tasks 1–4, choose a workflow.	"Choose a Workflow" on page 39-10

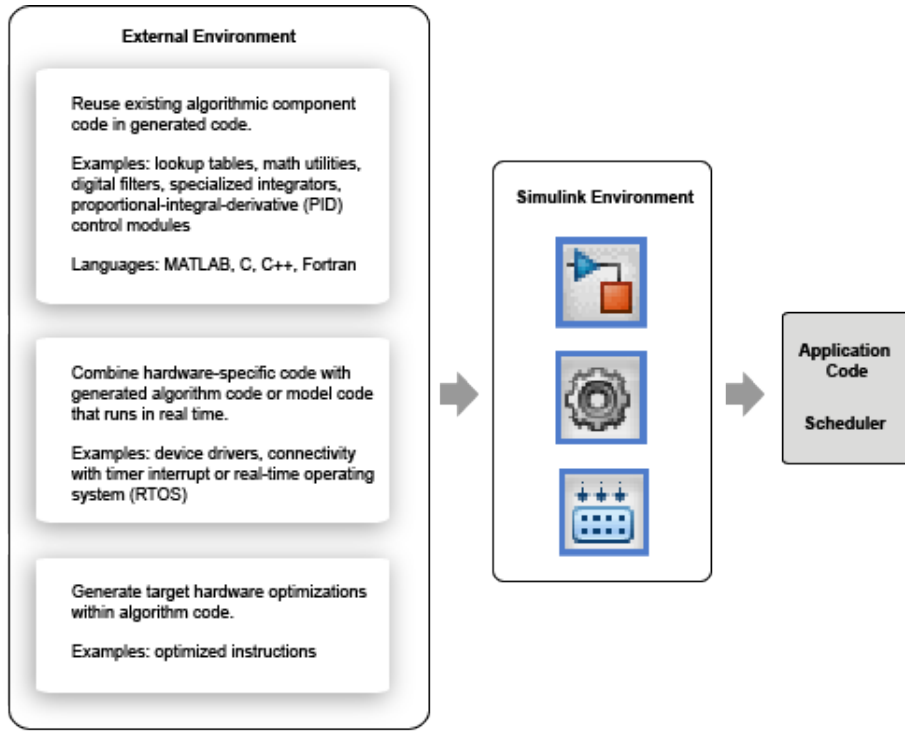
Choose a Software Execution Framework

The code generator supports two types of software execution frameworks—single top model and multiple top-level, as described in "Design Models for Generated Embedded

Code Deployment" on page 1-2. The first question to answer concerns which of the two frameworks applies to your project.

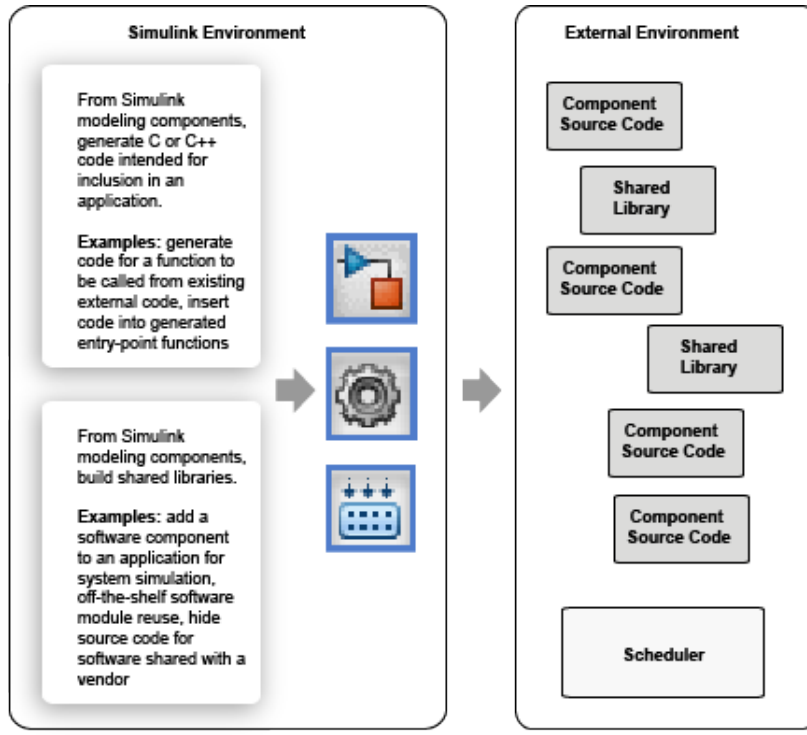
- Single top model

  Generate one set of application code files from external code and code that the Simulink C/C++ code generator produces. The generated code includes a scheduler. In this case, you *import* code into the Simulink code generation environment.



- Single top model or multiple top-level models

  Integrate C or C++ code that the code generator produces from model components with external application code and an external scheduler. You *export* generated code from the Simulink code generation environment.



Importing calls to external device driver code into a model and generating code for that model for export involves importing and exporting code.

Based on goals and requirements, external code integration is characterized in several ways, requiring different workflows and integration tooling:

- Import existing external code into generated code.
  - Call reusable external algorithm code for simulation and code generation.
  - Place external C/C++ code in generated code.
  - Call external device drivers.
  - Apply function and operator code replacements.
  - Interface with external timer interrupt or scheduler.
  - Generate replacement code for specific run-time environment.

- Export generated code for inclusion in external code base.
  - Generate component source code for export.
  - Generate shared library for export.

Next, see "Evaluate Characteristics of External Code" on page 39-7.

Evaluate Characteristics of External Code

Before choosing an external integration workflow, evaluate these characteristics of the external code. To interface with external code, generated C or C++ code handles one or more of the external code characteristics. An understanding of these characteristics and your requirements for modeling, simulation, and code generation helps you choose the optimal workflow for your integration scenario. (See "Identify Integration Requirements" on page 39-8.)

Characteristic	What to Consider
Hardware dependency	Is the external code hardware-dependent? Utility functions, lookup tables, and filters are examples of hardware-independent code. Device drivers interact directly with hardware. They depend on characteristics of the hardware. For example, a device driver for an analog-to-digital converter initializes, reads data from, and writes data to hardware registers. Hardware differences and dependencies concern data type size, endianess, shift operations, compiler directives, and optimized function and operator support. Other code interfaces with device drivers by using an API and data mapped to specific memory addresses. Typically, simulation on a development computer is not possible. Reading from and writing to a register during simulation on a development computer produces unexpected and unwanted results.
Reusable	Is the external code a reusable software module? Examples include utility functions, lookup tables, filters, specialized integrators, and proportional-integral-derivative (PID) control modules.
Dependency on data persistence between function calls	Does the external code require persistent data? For example, a call to a first order filter function uses the output of the previous call to the function to calculate a new output value. You have the option of defining the data as global or using shared memory outside the context of the function.

Characteristic	What to Consider
Data typing and interface	How complex is the data that the external code uses? What does the data interface look like? It consists of arguments, a return value, global variables, and access functions. What data types does the code use? Are the types limited to basic ANSI C integers, floating-point types, arrays of integers or floating-point types, and pointers to these types? Does the interface include structures or pointers to structures?
Fixed-point code	Is the external code designed to run on integer-only processors? If yes, the code exchanges and uses data represented as integers only. Data can be associated with fixed-point scaling or offsets.
External resource dependencies	Does the external code use data, functions, or macros defined outside the scope of the code? For example, the function can use a standard ANSI function, a shared library, or predefined constants. In these cases, you must inform the compiler and linker of the paths and file names of the external resources.
External solver required	Are you using the external function for advanced development or rapid prototyping to describe a system with a continuous transfer function or a set of differential equations? If yes, the external code relies on an external solver.

Next, see “Identify Integration Requirements” on page 39-8.

Identify Integration Requirements

Before choosing an external integration workflow, review these integration requirements. An understanding of these requirements and the characteristics of your external code helps you choose the optimal workflow for your integration scenario. (See “Evaluate Characteristics of External Code” on page 39-7.)

Requirement	What to Consider
Effort	What level of effort is planned for the integration project—low, medium, or high?
Learning effort	What is the programming experience of assigned project resources? How much experience do assigned resources have with Simulink and MathWorks C/C++ code generation products?

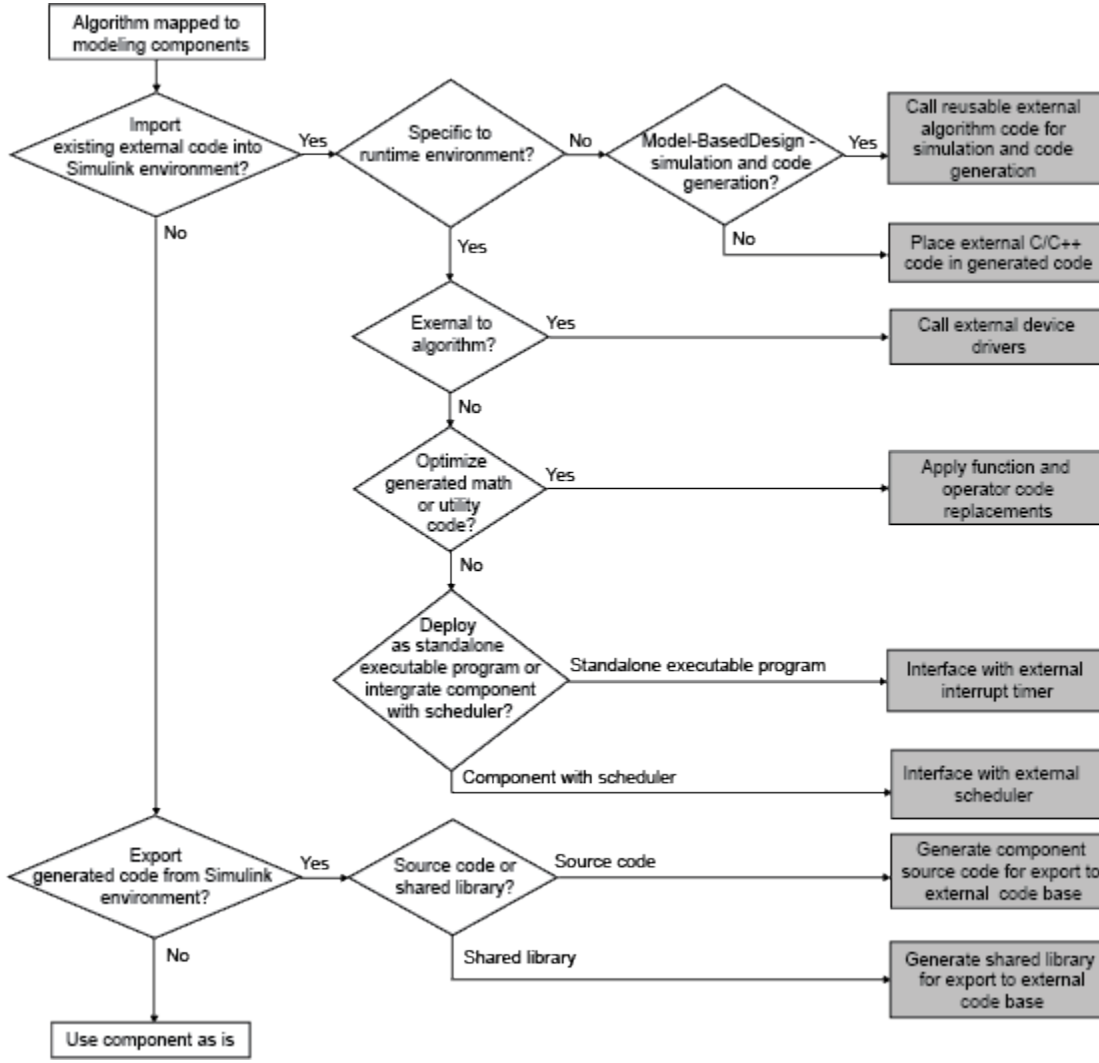
Requirement	What to Consider
Simulation and code generation behaviors	Do you want to take advantage of Model-Based Design? To take full advantage of Model-Based Design, convert code to modeling elements, which you can then use in the Simulink and Stateflow simulation environment. Then, simulate and generate code for the integrated component. Use software-in-the-loop (SIL) or processor-in-the-loop (PIL) testing to verify whether algorithm behavior is the same in both environments.
Data interface and typing	• Does your model or generated code need to exchange data with the external function? If so, map inputs, outputs, and parameters to the external function interface. Typical function interfaces involve function arguments and return values, global variables, and access functions, such as `getRPM`. • Do you want to represent arrays, structures, or enumerated types? In the Simulink environment, you can represent these types as vectors, buses, and `IntEnum`, respectively. • Is fixed-point support required? If you use the Simulink fixed-point interface, you can scale and specify offsets. • Does the external code use company-specific data types? If yes and you have Embedded Coder software, create alias types to represent those external types. The code generator uses the alias types in the code that it produces. For example, once defined, you can specify an alias type in a function prototype, for a temporary variable, or for block output. • Does the code exchange data with shared memory? If yes, define and use memory sections.
Direct function call	Do you want to call C external code directly from a model? You can choose from mechanisms, such as the Legacy Code Tool, Stateflow external code interface and chart action language, and the MATLAB Function block.
Insertion of external code into generated code	Do you want to control the placement of external code within generated code? Do you want to insert code into generated entry-point functions? You can place code within generated code by using model configuration parameters or custom code blocks.

Requirement	What to Consider
Code generation optimization support	Do you want to optimize the code that the code generator produces? If so, you can configure the model for the code generator to optimize the code it produces based on application objectives, such as execution, ROM, and RAM efficiency. You also have the option of using code replacement libraries.
Files required	Do you want to minimize the number of files that you maintain? Some external code integration tools require that you maintain separate files for defining simulation and code generation.

Next, see “Choose a Workflow” on page 39-10.

Choose a Workflow

To choose a workflow for each integration point, use the following flow diagram . The gray boxes identify common workflows and provide links to more information. Click the gray box that best addresses the requirements of an integration point.



More About

- "Call Reusable External Algorithm Code for Simulation and Code Generation" on page 39-13
- "Place External C/C++ Code in Generated Code" on page 39-27

Call Reusable External Algorithm Code for Simulation and Code Generation

Code reuse offers business and technological advantages. From a business perspective, code reuse saves time and resources. From a technological perspective, code reuse promotes consistency and reduces memory requirements. Other considerations include:

- Modularizing an application
- Reusing an optimized algorithm
- Interfacing with a predefined dataset
- Developing application variants

Examples of reusable hardware-independent algorithmic code to consider importing into the Simulink environment for simulation and code generation include:

- Utility functions
- Lookup tables
- Digital filters
- Specialized integrators
- Proportional-integral-derivative (PID) control modules

Workflow

To call reusable external algorithm code for simulation and code generation, iterate through the tasks listed in this table.

Task	Action	More Information
1	Review your assessment of external code characteristics and integration requirements.	“Choose an External Code Integration Workflow” on page 39-4
2	Based on the programming language of the external code, choose an integration approach to add the external code to a Simulink model.	“Choose an Integration Approach” on page 39-14

Task	Action	More Information
3	Verify algorithm behavior and performance by simulating the model.	"Simulation" (Simulink)
4	Define the representation of model data for code generation.	"Exchange Data Between External C/C++ Code and Simulink Model or Generated Code" on page 39-86
5	Configure the model for code generation.	"Generate Code That Matches Appearance of External Code" on page 39-95 and "Model Configuration"
6	Generate code and a code generation report.	"Code Generation"
7	Review the generated code interface and static code metrics.	"Analyze the Generated Code Interface" on page 35-21 and "Static Code Metrics" on page 35-34
8	Build an executable program from the model.	"Build Integrated Code Within the Simulink Environment" on page 39-41
9	Verify that executable program behaves as expected.	"Numerical Equivalence Testing"
10	Verify that executable program performs as expected.	"Code Execution Profiling"

Choose an Integration Approach

Several approaches are available for integrating reusable algorithmic code into the Simulink environment for code generation. Some approaches integrate external code directly. Other approaches convert the external code to Simulink or Stateflow modeling elements for simulation, and later for code generation from the modeled design. The integration approach that you choose depends on:

- Programming language of the external code — MATLAB, C, C++, or Fortran
- Your programming language experience and preference
- Performance requirements
- Whether the algorithm must model continuous time dynamics or you are integrating the algorithm into an application that uses discrete and continuous time

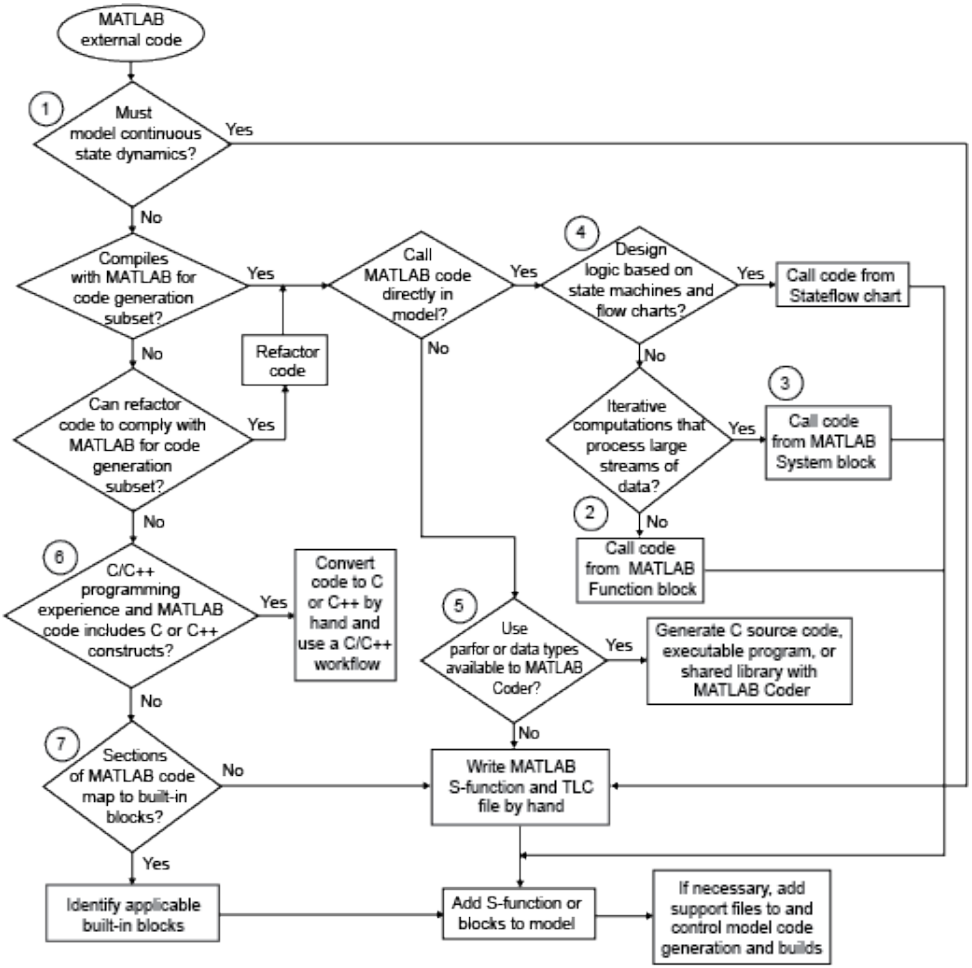
- Whether you want to take advantage of Model-Based Design
- Level of control required over the code that the code generator produces

To choose an approach for a reusable algorithm, see the subsection that matches the programming language of your external algorithm code.

- "Integration Approaches for External MATLAB Code" on page 39-15
- "Integration Approaches for External C or C++ Code" on page 39-18
- "Integration Approaches for External Fortran Code" on page 39-24

Integration Approaches for External MATLAB Code

Multiple approaches are available for integrating external MATLAB code into the Simulink environment. The following diagram and table help you choose the best integration approach for your application based on integration requirements.



	Condition or Requirement	Action	More Information
1	The algorithm must model continuous state dynamics.	Write a MATLAB S-function and, for generating code, a corresponding TLC file for the algorithm. Add the S-function to your model.	• "MATLAB S-Functions" (Simulink) • "Write S-Function and TLC Files By Hand" on page 11-66 • "Target Language Compiler" (Simulink Coder)

	Condition or Requirement	Action	More Information
2	External code complies with the MATLAB code for code generation subset and you want to call MATLAB code from a Simulink model.	Add a MATLAB Function block to the model. Embed the MATLAB code in that block.	• "Integrate C Code Using the MATLAB Function Block" (Simulink) • MATLAB Function
3	External code complies with the MATLAB code for code generation subset, you want to call MATLAB code from a Simulink model, and your algorithm includes iterative computations that process large streams of data.	Add a MATLAB System block to the model. Embed the MATLAB code in that block as a System object™.	• "Integrate C Code Using the MATLAB Function Block" (Simulink) • MATLAB System
4	External code complies with the MATLAB code for code generation subset, you want to call MATLAB code from a Simulink model, and your algorithm includes design logic that is based on state machines and flow charts.	Add a Stateflow chart to the model. Call the external code from the chart, using MATLAB as the action language.	• "Chart Programming Basics" (Stateflow) • "Insert External Code into Stateflow Charts" on page 39-24
5	You want to use the `parfor` function for parallel computing or interface data types that are available to MATLAB Coder but are not available to Simulink Coder or Embedded Coder.	Use MATLAB Coder software to generate C code. Then, call that generated code as external C code.	• "C Code Generation Using the MATLAB Coder App" (MATLAB Coder) • "Getting Started with MATLAB Coder" (MATLAB Coder)

	Condition or Requirement	Action	More Information
6	You have C or C ++ programming experience and the external MATLAB code is compact and primarily uses C or C++ constructs.	Manually convert the MATLAB code to C or C++ code. Choose an integration approach for C or C++ code.	"Integration Approaches for External C or C++ Code" on page 39-18
7	Sections of the external MATLAB code map to built-in blocks.	Develop the algorithm in the context of a model, using the applicable built-in blocks.	• "Model Editing Fundamentals" (Simulink) and "Component-Based Modeling" (Simulink) • "Supported Products and Block Usage" (Simulink Coder)

To embed external MATLAB code in a MATLAB Function block or generate C or C++ code from MATLAB code with the MATLAB Coder software, the MATLAB code must use functions and classes supported for C/C++ code generation.

- "Functions and Objects Supported for C/C++ Code Generation — Alphabetical List" (MATLAB Coder)
- "Functions and Objects Supported for C/C++ Code Generation — Category List" (MATLAB Coder)

Integration Approaches for External C or C++ Code

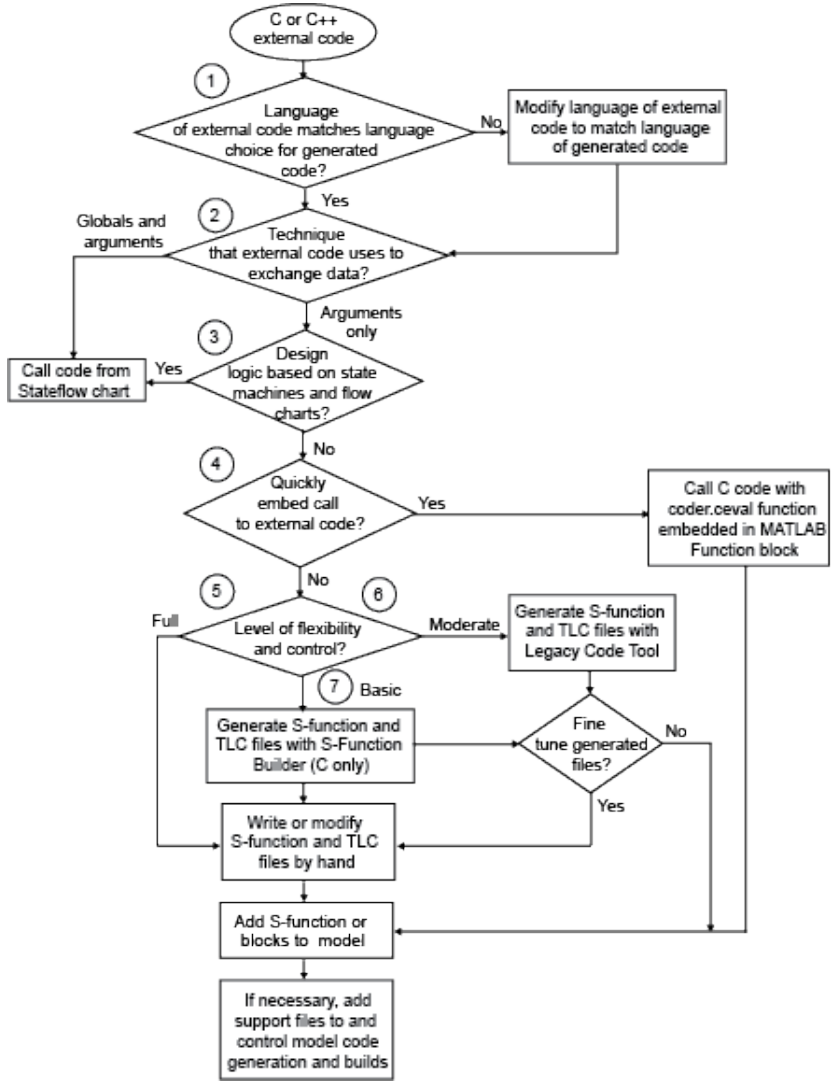
Under most circumstances, you can integrate external code written in C or C++ into the Simulink environment by generating S-functions and TLC files with the Legacy Code Tool. This tool uses specifications that you supply as MATLAB code to transform existing MATLAB functions into C MEX S-functions that you can include in Simulink models and call from generated code. For details, see "Legacy Code Integration" (Simulink) and "Import Calls to External Code into Generated Code with Legacy Code Tool" on page 11-7.

In comparison to alternative approaches, Legacy Code Tool is the easiest to use and generates code optimized enough for embedded systems. Consider alternative approaches if one or more of the following conditions exist:

- The external code uses global variables to exchange data.
- Programming experience is limited.

- The algorithm must model discrete and continuous state dynamics.
- You want to include the integrated external code in a Stateflow chart.
- The external code requires a fixed-point interface.
- You want maximum flexibility for controlling what code the code generator produces.
- You quickly want to embed a call to the external code in a call to the `coder.ceval` function that is embedded in the MATLAB Function block, and performance is not an issue.

This diagram and table help you choose the best integration approach based on your integration requirements.



	Condition or Requirement	Action	More Information
1	You want to integrate external C code with generated C++ code or conversely	Match the language choice for the generated code by modifying the	"Modify Programming Language of External Code to Match Generated Code" on page 39-23

	Condition or Requirement	Action	More Information
		language of the external code.	
2	Your algorithm includes design logic that is based on state machines and flow charts. Or, a function that you want to integrate must exchange data with a model by using global variables. The function defines the global variables and uses them to write output rather than returning a value (return) or writing output to an argument.	Add a Stateflow chart to the model. Call the external code from the chart, using C as the action language. In the chart, write code that calls the external function and reads from and writes to the global variables. To perform calculations with output of the external code, the model must read from the global variable during execution.	"Insert External Code into Stateflow Charts" on page 39-24
3	You want to include external C or C++ code in a Stateflow chart for simulation and code generation.	Configure the model that contains the chart to apply the external C or C ++ code.	• "Custom Code Algorithm" (Stateflow) • "Call C Functions in C Charts" (Stateflow) • "Insert External Code into Stateflow Charts" on page 39-24
4	You quickly want to embed a call to external C or C++ code in a model. Performance is not an issue.	Call the C or C++ code with the `coder.ceval` function from within a MATLAB Function block.	• `coder.ceval` function • "Integrate C Code Using the MATLAB Function Block" (Simulink) • MATLAB Function block

	Condition or Requirement	Action	More Information
5	The application requires more entry-point functions than the code generator typically produces —for example, more than *model*_step, *model*_initialize, and *model*_terminate. You want maximum flexibility for controlling what code the code generator produces.	Manually write an S-function and TLC file.	• "S-Function Basics" (Simulink) • "S-Functions and Code Generation" on page 11-2 • "C S-Function Examples" (Simulink) and "C++ S-Function Examples" (Simulink)
6	You want to simulate and generate external code for a discrete time application. Optimizing generated code is essential. You want ease of use with moderate flexibility for controlling what code the code generator produces. You have C or C++ programming experience, but you prefer to generate the files for adding the code to a model.	Generate S-function and TLC files by using the Legacy Code Tool. If necessary, refine the generated code manually to meet application requirements. (If you change the generated code, you lose the changes if you regenerate the S-function and TLC files.)	• "Integrate C Functions Using Legacy Code Tool" (Simulink) • "Import Calls to External Code into Generated Code with Legacy Code Tool" (Simulink Coder)

	Condition or Requirement	Action	More Information
7	The algorithm must model discrete and continuous state dynamics for simulation and rapid prototyping. The external code requires a fixed-point interface. Programming experience is limited. You want ease of use with basic flexibility for controlling what code the code generator produces for rapid prototyping.	Generate S-function and TLC files by using the S-Function Builder. If necessary, refine the generated code manually to meet application requirements. (If you change the generated code, you lose the changes if you regenerate the S-function and TLC files.)	• "Build S-Functions Automatically" (Simulink) • "Automate S-Function Generation with S-Function Builder" on page 11-61

Modify Programming Language of External Code to Match Generated Code

To integrate external C code with generated C++ code or conversely, modify the language of the external code to match the programming language choice for the generated code. Options for making the programming language match include:

- Writing or rewriting the external code in the language choice for the generated code.
- If you are generating C++ code and the external code is C code, for each C function, create a header file that prototypes the function. Use this format:

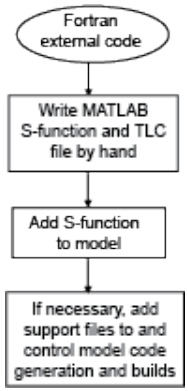
  ```
  #ifdef __cplusplus
  extern "C" {
  #endif
  int my_c_function_wrapper();
  #ifdef __cplusplus
  }
  #endif
  ```

  The prototype serves as a function wrapper. If your compiler supports C++ code, the value `__cplusplus` is defined. The linkage specification `extern "C"` specifies C linkage without name mangling.
- If you are generating C code and the external code is C++ code, include an `extern "C"` linkage specification in each `.cpp` file. For example, the following example shows C++ code in the file `my_func.cpp`:

```
extern "C" {

int my_cpp_function()
{
  ...
}
}
```

Integration Approaches for External Fortran Code

To integrate external Fortran code, write an S-function and corresponding TLC file.



See “S-Function Basics” (Simulink), “Fortran S-Functions” (Simulink), “S-Functions and Code Generation” on page 11-2, and“Fortran S-Function Examples” (Simulink).

Insert External Code into Stateflow Charts

Integrate External Code for Library Charts

To integrate external code that applies only to Stateflow library charts for code generation, for each library model that contributes a chart to your main model, complete these steps. Then, generate code.

1 In the Stateflow Editor, select **Code > C/C++ Code > Code Generation Options**.

2. In the Model Configuration Parameters dialog box, select **Code Generation** > **Use local custom code settings (do not inherit from main model)**.

   The library model retains its own custom code settings during code generation.

3. Specify your custom code in the subpanes.

   Follow the guidelines in "Specify Relative Paths for Custom Code" (Stateflow).

   If you specified custom code settings for simulation, you can apply these settings to code generation. To avoid entering the same information twice, select **Use the same custom code settings as Simulation Target**.

4. Click **OK**.

After completing these steps for each library model, generate code.

Integrate External Code for All Charts

To integrate external code that applies to all charts for code generation:

1. Specify custom code options for code generation of your main model.

   a. In the Model Configuration Parameters dialog box, select **Code Generation** > **Custom Code**.

   b. In the custom code text fields, specify your custom code.

      Follow the guidelines in "Specify Relative Paths for Custom Code" (Stateflow).

      If you specified custom code settings for simulation, you can apply these settings to code generation. To avoid entering the same information twice, select **Use the same custom code settings as Simulation Target**.

2. Configure code generation for each library model that contributes a chart to your main model.

   a. In the Stateflow Editor, select **Code** > **C/C++ Code** > **Code Generation Options**.

   b. In the **Code Generation** pane, clear the **Use local custom code settings (do not inherit from main model)** check box.

      The library charts inherit the custom code settings of your main model.

   c. Click **OK**.

3 Generate code.

More About

- “Integrate C Functions Using Legacy Code Tool” (Simulink)
- “Import Calls to External Code into Generated Code with Legacy Code Tool” (Simulink Coder)
- “Custom Code Algorithm” (Stateflow)
- “Integrate C Code Using the MATLAB Function Block” (Simulink)
- “S-Function Basics” (Simulink)
- “S-Functions and Code Generation” on page 11-2

Place External C/C++ Code in Generated Code

In this section...

You can customize code that the code generator produces for a model by specifying external code with custom code blocks or model configuration parameters.

- Place code at the start and end of the generated code for the root model.
- Place declaration, body, and exit code in generated function code for blocks in the root model or nonvirtual subsystems.

The functions that you can augment with external code depends on the functions that the code generator produces for blocks that are in the model. For example, if a model or atomic subsystem includes blocks that have states, you can specify code for a disable function. Likewise, if you need the code for a block to save data, free memory, or reset target hardware, specify code for a terminate function. For more information, see "Block Target File Methods" (Simulink Coder).

Workflow

To place external C or C++ code at specific locations in code that the code generator produces for root models and subsystems, iterate through the tasks listed in this table.

Task	Action	More Information
1	If you want to integrate external C code with generated C++ code or conversely, modify the language of the external code to match the language choice for the generated code.	"Modify Programming Language of External Code to Match Generated Code" (Simulink Coder)

Task	Action	More Information
2	Review your assessment of external code characteristics and integration requirements.	"Choose an External Code Integration Workflow" on page 39-4
3	If necessary, rewrite code in C or C++.	
4	Choose an integration approach to add the external code to a Simulink model.	"Choose an Integration Approach" on page 39-28
5	Define the representation of model data for code generation.	"Exchange Data Between External C/C++ Code and Simulink Model or Generated Code" on page 39-86
6	Configure the model for code generation.	"Generate Code That Matches Appearance of External Code" on page 39-95 and "Model Configuration"
7	Generate code and a code generation report.	"Code Generation"
8	Review the generated code interface and static code metrics.	"Analyze the Generated Code Interface" on page 35-21 and "Static Code Metrics" on page 35-34
9	Build an executable program from the model.	"Build Integrated Code Within the Simulink Environment" on page 39-41
10	Verify that executable program performs as expected.	"Numerical Equivalence Testing" and "Code Execution Profiling"

Choose an Integration Approach

Within the Simulink modeling environment, two approaches are available for placing external C or C++ code into sections of code that the code generator produces:

- Add blocks from the Custom Code library to a root model or atomic subsystem.
- Set model configuration parameters on the **Code Generation** > **Custom Code** pane.

The following table compares the two approaches. Choose the approach that aligns best with your integration requirements. For more information about how to apply each approach, see "Integrate External Code by Using Custom Code Blocks" on page

39-29 and "Integrate External Code by Using Model Configuration Parameters" on page 39-32.

Requirement	Blocks	Model Configuration Parameters
Include a representation of your external code in the modeling canvas	✓	
Place code in functions generated for root models	✓	✓
Place code in functions generated for atomic subsystems	✓	
Save code placement in a model configuration set		✓
Place code at the top and bottom of the header and source files generated for a model	✓	✓
Place code within declaration, execution, and exit sections of the `SystemInitialize` and`SystemTerminate` functions that the code generator creates	✓	✓
Place code within declaration, execution, and exit sections of the `SystemStart`, `SystemEnable`, `SystemDisable`, `SystemOutputs`, `SystemUpdate`, or`SystemDerivatives` functions that the code generator creates	✓	
Add preprocessor macro definitions to generated code		✓
Use the custom code settings that are specified for the simulation target		✓
Configure a library model to use custom code settings of the parent model to which the library is linked		✓

Integrate External Code by Using Custom Code Blocks

• "Add External Code to Generated Start Function" on page 39-31

Custom Code Block Library

The Custom Code block library contains blocks that you can use to place external C or C++ code into specific locations and functions within code that the code generator produces. The library consists of 10 blocks that add your code to the model header (*model*.h) and source (*model*.c or *model*.cpp) files that the code generator produces.

The Model Header and Model Source blocks add external code at the top and bottom of header and source files that the code generator produces for a root model. Theses blocks display two text fields into which you can type or paste code. One field specifies code that you want to place at the top of the generated header or source file. The second field specifies code that you want to place at the bottom of the file.

The remaining blocks add external code to functions that the code generator produces for the root model or atomic subsystem that contains the block. The blocks display text fields into which you can type or paste code that customizes functions that the code generator produces. The text fields correspond to the declaration, execution, and exit sections of code for a given function.

To Customize Code That	Use This Block
Computes continuous states	System Derivatives
Disables state	System Disable
Enables state	System Enable
Resets state	System Initialize
Produces output	System Outputs
Executes once	System Start
Saves data, free memory, reset target hardware	System Terminate
Requires updates at each major time step	System Update

The block and its location within a model determines where the code generator places the external code. For example, if the System Outputs block is at the root model level, the code generator places the code in the model `Outputs` function. If the block resides in a triggered or enabled subsystem, the code generator places the code in the subsystem `Outputs` function.

If the code generator does not need to generate a function that corresponds to a Custom Code block that you include in a model, the code generator does one of the following:

- Omits the external code that you specify in the Custom Code block.
- Returns an error, indicating that the model does not include a relevant block. In this case, remove the Custom Code block from the mode.

For more information, see "Block Target File Methods" (Simulink Coder).

Add Custom Code Blocks to the Modeling Canvas

To add the Custom Code library blocks to a model:

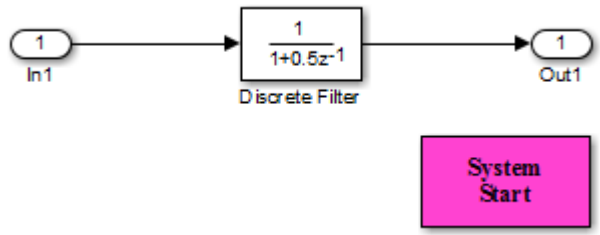
1. In the Simulink Library Browser, open the Custom Code block library. You can gain access to the library by:
   - Navigating to **Simulink Coder** > **Custom Code** in the browser.
   - Entering the MATLAB command `custcode`.
2. Drag the blocks that you want into your model or subsystem. Drag Model Header and Model Source blocks into root models only. Drag function-based Custom Code blocks into root models or atomic subsystems.

You can use models that contain Custom Code blocks as referenced models. The code generator ignores the blocks when producing code for a simulation target. When producing code for a code generation target, the code generator includes and compiles the custom code.

Add External Code to Generated Start Function

This example shows how to use the System Start block to place external C code in the code that the code generator produces for a model that includes a discrete filter.

1. Create the following model.



2. Configure the model for code generation.
3. Double-click the System Start block.
4. In the block parameters dialog box, in the **System Start Function Declaration Code** field, enter this code:

   ```
   unsigned int *ptr = 0xFFEE;
   ```

5. In the **System Start Function Execution Code** field, enter this code:

   ```
   /* Initialize hardware */
   *ptr = 0;
   ```

6. Click **OK**.
7. Generate code and a code generation report.
8. View the generated *model*`.c` file. Search for the string `start function`. You should find the following code, which includes the external code that you entered in steps 4 and 5.

   ```
   {
     {
       /* user code (Start function Header) */
       /* System '<Root>' */
       unsigned int *ptr = 0xFFEE;

       /* user code (Start function Body) */
       /* System '<Root>' */
       /* Initialize hardware */
       *ptr = 0;
     }
   }
   ```

For another example, see “Integrate External C Code Into Generated Code By Using Custom Code Blocks and Model Configuration Parameters” on page 39-34.

Integrate External Code by Using Model Configuration Parameters

Model configuration parameters provide a way to place external C or C++ code into specific locations and functions within code that the code generator produces.

To	Select
Insert external code near the top of the generated *model*`.c` or *model*`.cpp` file	**Source file**, and enter the external code to insert. **Note:** If you generate subsystem code into separate files, that code does not have access to external code that you specify with the **Source file**

To	Select
	parameter. For example, if you specify an include file as a **Source file** setting, the code generator inserts the `#include` near the top of the *`model`*`.c` or *`model`*`.cpp` file. The subsystem code that the code generator places in a separate file does not have access to declarations inside your included file. In this case, consider specifying your external code with the **Header file** parameter.
Insert external code near the top of the generated *`model`*`.h` file	**Header file**, and enter the external code to insert.
Insert external code inside the model initialize function in the *`model`*`.c` or *`model`*`.cpp` file	**Initialize function**, and enter the external code to insert.
Insert external code inside the model terminate function in the *`model`*`.c` or *`model`*`.cpp` file	**Terminate function**, and enter the external code to insert. Also select the **Terminate function required** parameter on the **Interface** pane.
Add preprocessor macro definitions	**Defines**, and enter a space-separated list of preprocessor macro definitions to add to the generated code. The list can include simple definitions (for example, `-DEF1`) and definitions with a value (for example, `-DDEF2=1`). Definitions can omit the `-D` (for example, `-DFOO=1` and `FOO=1` are equivalent). If a definition includes `-D`, the toolchain can override the flag if the toolchain uses a different flag for defines.
Use the same custom code parameter settings as the settings specified for simulation of MATLAB Function blocks, Stateflow charts, and Truth Table blocks	**Use the same custom code settings as Simulation Target** This parameter refers to the **Simulation Target** pane in the Configuration Parameters dialog box.

To	Select
Enable a library model to use custom code settings unique from the parent model to which the library is linked	**Use local custom code settings (do not inherit from main model)** This parameter is available only for library models that contain MATLAB Function blocks, Stateflow charts, or Truth Table blocks.

To include a header file in an external header file, add `#ifndef` code. Using this code avoids multiple inclusions. For example, in `rtwtypes.h`, the following `#include` guards are added:

```
#ifndef RTW_HEADER_rtwtypes_h_
#define RTW_HEADER_rtwtypes_h_
...
#endif /* RTW_HEADER_rtwtypes_h_ */
```

For more information on how to add files names and locations of header, source, and shared library files to the build process, see "Build Integrated Code Within the Simulink Environment" on page 39-41.

Note The code generator includes external code that you include in a configuration set when generating code for software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulations. However, the code generator ignores external code that you include in a configuration set when producing code with the S-function, rapid simulation, or simulation system target file.

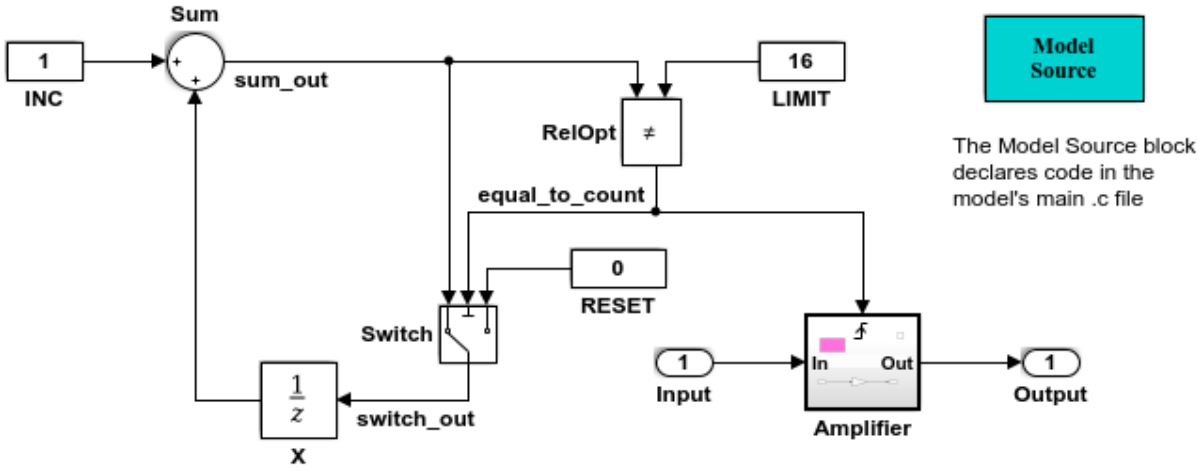
For more information about **Custom Code** parameters, see "Model Configuration Parameters: Code Generation Custom Code" (Simulink Coder). For an example, see "Integrate External C Code Into Generated Code By Using Custom Code Blocks and Model Configuration Parameters" on page 39-34.

Integrate External C Code Into Generated Code By Using Custom Code Blocks and Model Configuration Parameters

This example shows how to place external code in generated code by using custom code blocks and model configuration parameters.

1. Open the model `rtwdemo_slcustcode`.

```
open_system('rtwdemo_slcustcode')
```



Several techniques exist for incorporating custom code into Simulink Coder. This model shows the use of the Simulink Coder custom code blocks and the Configuration Parameters Code Generation Custom Code page:

1. The Model Source custom code block declares an integer GLOBAL_INT1 in <model>.c.
2. The Subsystem Outputs custom code block (inside subsystem Amplifier) uses GLOBAL_INT1.
3. The variable GLOBAL_INT2 is declared and set from the Configuration Parameters Code Generation Custom Code page, from the "Source file" and "Initialize function," respectively.

Some overlap exists between custom code blocks and custom code specified using configuration parameters, but custom code blocks provide much finer granularity of code placement, and have the advantage of being graphical.

Generate Code Using Simulink Coder (double-click) | **Generate Code Using Embedded Coder (double-click)** | **View Custom Code Configuration (double-click)** | **View Custom Code Library (double-click)**

2. Open the Model Configuration Parameters dialog box and navigate to the **Custom Code** pane.

3. Examine the settings for parameters **Source file** and **Initialize function**.

- **Source file** specifies a comment and sets the variable `GLOBAL_INT2` to -1.
- **Initialize function** intializes the variable `GLOBAL_INT2` to 1.

4. Close the dialog box.

5. Double-click the Model Source block. The **Top of Model Source** field specifies that the code generator declare the variable `GLOBAL_INT1` and set it to 0 at the top of the generated file `rtwdemo_slcustcode.c`.

6. Open the triggered subsystem `Amplifier`. The subsystem includes the System Outputs block. The code generator places code that you specify in that block in the generated code for the nearest parent atomic subsystem. In this case, the code generator places the external code in the generated code for the `Amplifier` subsystem. The external code:

- Declares the pointer variable `*intPtr` and intitializes it with the value of variable `GLOBAL_INT1`.
- Sets the pointer variable to -1 during execution.
- Resets the pointer variable to 0 before exiting.

7. Generate code and a code generation report.

8. Examine the code in the generated source file `rtwdemo_slcustcode.c`. At the top of the file, after the `#include` statements, you find the following declaration code. The example specifies the first declaration with the **Source file** configuration parameter and the second declaration with the Model Source block.

```
int_T GLOBAL_INT2 = -1;

int_T GLOBAL_INT1 = 0;
```

The Output function for the `Amplifier` subsystem includes the following code, which shows the external code integrated with generated code that applies the gain. The example specifies the three lines of code for the pointer variable with the System Output block in the `Amplifier` subsystem.

```
int_T *intPtr = &GLOBAL_INT1;
```

```
*intPtr = -1;

rtwdemo_slcustcode_Y.Output = rtwdemo_slcustcode_U.Input << 1;

*intPtr = 0;
```

The following assignment appears in the model initialize entry-point function. The example specifies this assignment with the **Initialize function** configuration parameter.

```
GLOBAL_INT2 = 1;
```

More About

- “Configure a Model for Code Generation” on page 2-2

Call External Device Drivers

Device drivers for protocols and target hardware are essential to many real-time development projects. For example, you can have a working device driver that you want to integrate with algorithmic code that has to read data from and write data to the I/O device that the driver supports. The code generator can produce a single set of application source files from an algorithm model and integrated driver code written in C or C++.

To call external device driver code from the Simulink environment, iterate through the tasks in this table.

Task	Action	More Information
1	Review your assessment of external code characteristics and integration requirements.	"Choose an External Code Integration Workflow" on page 39-4
2	Define the representation of model data for code generation.	"Exchange Data Between External C/C++ Code and Simulink Model or Generated Code" on page 39-86
3	Generate S-function and TLC files by using the Legacy Code Tool. If necessary, refine the generated code manually to meet application requirements.	• "Integrate C Functions Using Legacy Code Tool" (Simulink) • "Import Calls to External Code into Generated Code with Legacy Code Tool" (Simulink Coder)
4	Verify algorithm behavior and performance by simulating the model.	"Simulation" (Simulink)
5	Configure the model for code generation.	"Generate Code That Matches Appearance of External Code" on page 39-95 and "Model Configuration"
6	Generate code and a code generation report.	"Code Generation"
7	Review the generated code interface and static code metrics.	"Analyze the Generated Code Interface" on page 35-21 and "Static Code Metrics" on page 35-34
8	Build an executable program from the model.	"Build Integrated Code Within the Simulink Environment" on page 39-41

Task	Action	More Information
9	Verify that executable program behaves and performs as expected.	“Numerical Equivalence Testing”
10	Verify that executable program performs as expected.	“Code Execution Profiling”

More About

- “Integrate C Functions Using Legacy Code Tool” (Simulink)
- “Import Calls to External Code into Generated Code with Legacy Code Tool” (Simulink Coder)
- “About Embedded Target Development” (Simulink Coder)

Apply Function and Operator Code Replacements

If your generated code must use functions and operators that are consistent with external code, configure the code generator to use a code replacement library (CRL). By default, the code generator does not apply a code replacement library. You can choose from several libraries that MathWorks provides, including GNU C99 extensions, AUTOSAR 4.0, and several Intel platform-specific IPP and IPP/SSE libraries. Depending on the products that you have, other libraries might be available. If you have Embedded Coder software, you can view and choose from additional libraries and you can create custom code replacement libraries.

More About

- "What Is Code Replacement?" on page 37-2
- "What Is Code Replacement Customization?" on page 51-3
- Code Replacement Viewer
- Code Replacement Tool

Build Integrated Code Within the Simulink Environment

Workflow

To build executable programs that integrate generated code and external C or C++ code, iterate through the tasks in this table.

Task	Action	More Information
1	Choose whether to use the toolchain approach or template makefile approach build process.	"Choose and Configure Build Process" on page 40-14 For an example, see "Build Process Workflow for a Real-Time STF" on page 40-30.
2	Configuredbuild process support for your external code.	"Configure Parameters for Integrated Code Build Process" on page 39-42
3	Configure S-Function build support for your external code.	"Build Support for S-Functions" on page 39-44 "Use makecfg to Customize Generated Makefiles for S-Functions" on page 70-24 For examples, see "Call External C Code from Model and Generated Code" and "Call Reusable External Algorithm Code for Simulation and Code Generation" on page 39-13.
4	Configure build process to find the external code source, library, and header files.	"Manage Build Process File Dependencies" on page 33-52 "Control Library Location and Naming During Build" on page 70-7
5	Set up custom build processing required for your external code integration.	For the build process customization workflow, see "Customize Post-Code-Generation Build Processing" on page 70-14.

Task	Action	More Information
		To automate applying build customizations to a toolchain approach build, see "Customize Build Process with sl_customization.m" on page 70-38. To automate applying build customizations to a template makefile approach build, see "Customize Build Process with STF_make_rtw_hook File" on page 70-31.

Configure Parameters for Integrated Code Build Process

The table provides a guide to configuration parameters that support the build process for external code integration. For information about folders for your external code, see "Manage Build Process Folders" on page 33-37. If you choose to place your external code in the "Code generation folder" (Simulink), see "Preserve External Code Files in Build Folder" on page 39-43.

To	Select
Add include folders, which contain header files, to the build process	**Configuration Parameters > Code Generation > Custom Code > Additional build information > Include directories**, and enter the absolute or relative paths to the folders. If you specify relative paths, the paths must be relative to the folder containing your model files, not relative to the build folder. The order in which you specify the folders is the order in which they are searched for header, source, and library files.
Add source files to be compiled and linked	**Configuration Parameters > Code Generation > Custom Code > Additional build information > Source files**, and enter the full paths or just the file names for the files. Enter just the file name if the file is in the current MATLAB folder or in one of the include folders. For each additional

To	Select

	source that you specify, the build process expands a generic rule in the template makefile for the folder in which the source file is located. For example, if a source file is located in folder `inc`, the build process adds a rule similar to the following:
	
	`%.obj: buildir\inc\%.c`
	`$(CC) -c -Fo$(@F) $(CFLAGS) $<`
	
	The build process adds the rules in the order that you list the source files.

Add libraries to be linked	**Configuration Parameters > Code Generation > Custom Code > Additional build information > Libraries**, and enter the full paths or just the file names for the libraries. Enter just the file name if the library is located in the current MATLAB folder or in one of the include folders.
Use the same custom code settings as those specified for simulation of MATLAB Function blocks, Stateflow charts, and Truth Table blocks	**Configuration Parameters > All Parameters > Code Generation > Custom Code > Use the same custom code settings as Simulation Target** **Note** This parameter refers to the **Simulation Target** pane in the Configuration Parameters dialog box.
Enable a library model to use custom code settings unique from the parent model to which the library is linked	**Configuration Parameters > All Parameters > Code Generation > Custom Code > Use local custom code settings (do not inherit from main model)** **Note** This parameter is available only for library models that contain MATLAB Function blocks, Stateflow charts, or Truth Table blocks.

Preserve External Code Files in Build Folder

By default, the build process deletes foreign source files. You can preserve foreign source files by following these guidelines.

If you put a `.c/.cpp` or `.h` source file in a build folder, and you want to prevent the code generator from deleting it during the TLC code generation process, insert the text `target specific file` in the first line of the `.c/.cpp` or `.h` file. For example:

```
/*  COMPANY-NAME target specific file
 *
 *   This file is created for use with the
 *   COMPANY-NAME target.
 *   It is used for ...
 */
...
```

Make sure that you spell the text "target specific file" as shown in the preceding example, and that the text is in the first line of the source file. Other text can appear before or after this text.

Flagging user files in this manner prevents postprocessing these files to indent them with generated source files. Auto-indenting occurred in previous releases to build folder files with names having the pattern *`model`*`_*.c/.cpp` (where `*` was text). The indenting is harmless, but can cause differences detected by source control software that can potentially trigger unnecessary updates.

Build Support for S-Functions

User-written S-Function blocks provide a powerful way to incorporate external code into the Simulink development environment. In most cases, you use S-functions to integrate existing external code with generated code. Several approaches to writing S-functions are available:

- "Write Noninlined S-Function and TLC Files" on page 11-66
- "Write Wrapper S-Function and TLC Files" on page 11-68
- "Write Fully Inlined S-Functions" on page 11-77
- "Write Fully Inlined S-Functions with mdlRTW Routine" on page 11-78
- "S-Functions That Support Code Reuse" on page 11-114
- "S-Functions for Multirate Multitasking Environments" on page 11-115

S-functions also provide the most flexible and capable way of including build information for legacy and custom code files in the build process.

There are different ways of adding S-functions to the build process.

Implicit Build Support

When building models with S-functions, the build process adds rules, include paths, and source file names to the generated makefile. The source files (`.h`, `.c`, and `.cpp`) for the S-function must be in the same folder as the S-function MEX-file. Whether using the toolchain approach or template makefile approach for builds, the build process propagates this information through the toolchain or template makefile.

- If the file *`sfcnname`*`.h` exists in the same folder as the S-function MEX-file (for example, *`sfcnname.mexext`*), the folder is added to the include path.
- If the file *`sfcnname`*`.c` or *`sfcnname`*`.cpp` exists in the same folder as the S-function MEX-file, the build process adds a makefile rule for compiling files from that folder.
- When an S-function is not inlined with a TLC file, the build process must compile the S-function source file. To determine the name of the source file to add to the list of files to compile, the build process searches for *`sfcnname`*`.cpp` on the MATLAB path. If the source file is found, the build process adds the source file name to the makefile. If *`sfcnname`*`.cpp` is not found on the path, the build process adds the file name *`sfcnname`*`.c` to the makefile, whether or not it is on the MATLAB path.

Note: For the Simulink engine to find the MEX-file for simulation and code generation, it must exist on the MATLAB path or exist in our current MATLAB working folder.

Specify Additional Source Files for an S-Function

If your S-function has additional source file dependencies, you must add the names of the additional modules to the build process. Specify the file names:

- In the **S-function modules** field in the S-Function block parameter dialog box
- With the `SFunctionModules` parameter in a call to the `set_param` function

For example, suppose you build your S-function with multiple modules.

```
mex sfun_main.c sfun_module1.c sfun_module2.c
```

You can then add the modules to the build process by doing one of the following:

- In the S-function block dialog box, specify `sfun_main`, `sfun_module1`, and `sfun_module2` in the **S-function modules** field.
- At the MATLAB command prompt, enter:

```
set_param(sfun_block,'SFunctionModules','sfun_module1 sfun_module2')
```

Alternatively, you can define a variable to represent the parameter value.

```
modules = 'sfun_module1 sfun_module2'
          set_param(sfun_block,'SFunctionModules', modules)
```

The **S-function modules** field and `SFunctionModules` parameter do not support complete source file path specifications. To use the parameter, the code generator must find the additional source files when executing the makefile. For the code generator to locate the additional files, place them in the same folder as the S-function MEX-file. You can then leverage the implicit build support described in "Implicit Build Support" on page 39-45.

When you are ready to generate code, force the code generator to rebuild the top model, as described in "Control Regeneration of Top Model Code" (Simulink Coder).

For more complicated S-function file dependencies, such as specifying source files in other locations or specifying libraries or object files, use the `rtwmakecfg.m` API, as described in "Use rtwmakecfg.m API to Customize Generated Makefiles" on page 70-26.

Use TLC Library Functions

If you inline your S-function by writing a TLC file, you can add source file names to the build process by using the TLC library function `LibAddToModelSources`. For details, see "LibAddSourceFileCustomSection(file, builtInSection, newSection)" (Simulink Coder).

Note: This function does not support complete source file path specifications. The function assumes that the code generator can find the additional source files when executing the makefile.

Another useful TLC library function is `LibAddToCommonIncludes`. Use this function in a `#include` statement to include S-function header files in the generated *`model`*`.h` header file. For details, see "LibAddToCommonIncludes(incFileName)" (Simulink Coder).

For more complicated S-function file dependencies, such as specifying source files in other locations or specifying libraries or object files, use the `rtwmakecfg.m` API, as described in "Use rtwmakecfg.m API to Customize Generated Makefiles" on page 70-26.

Precompile S-Function Libraries

You can precompile new or updated S-function libraries (MEX-files) for a model by using the MATLAB language function `rtw_precompile_libs`. Using a specified model and a library build specification, this function builds and places the libraries in a precompiled library folder.

By precompiling S-function libraries, you can optimize system builds. Once your precompiled libraries exist, the build process can omit library compilation from subsequent builds. For models that use numerous libraries, the time savings for build processing can be significant.

To use `rtw_precompile_libs`:

1. Set the library file suffix, including the file type extension, based on your system platform.

   Consider determining the type of platform, and then use the `TargetLibSuffix` parameter to set the library suffix accordingly. For example, when applying a suffix for a GRT target, you can set the suffix to `_std.a` for a UNIX platform and `_vcx64.lib` for a Windows platform.

   ```
   if isunix
     suffix = '_std.a';
   else
     suffix = '_vcx64.lib';
   end

   set_param(my_model,'TargetLibSuffix', suffix);
   ```

   There are a number of factors that influence the precompiled library suffix and extension. The following table provides examples for typical selections of system target file, the compiler toolchain, and other options that affect your choice of suffix and extension. For more information, examine the template make files in the `matlab/rtw/c/grt` folder or `matlab/rtw/c/ert` folder.

TMF File	COMPILER _TOOL_CHAIN Value	Precompiler Libraries (PRECOMP_LIBRARIES)			
		Library Suffix S-Function (EXPAND _LIBRARY _NAME Value)	Library Suffix Integer- Only Code (EXPAND _LIBRARY _NAME Value)	Library Suffix Optimize for Speed (EXPAND _LIBRARY _NAME Value)	Library Extension (EXPAND _LIBRARY _NAME Value)
ert_lcc64.tmf	lcc	_rtwsfcn_lcc	_int_ert_lcc	_ert_lcc	.lib

TMF File	COMPILER_TOOL_CHAIN Value	Precompiler Libraries (PRECOMP_LIBRARIES)			
		Library Suffix S-Function (EXPAND_LIBRARY_NAME Value)	Library Suffix Integer- Only Code (EXPAND_LIBRARY_NAME Value)	Library Suffix Optimize for Speed (EXPAND_LIBRARY_NAME Value)	Library Extension (EXPAND_LIBRARY_NAME Value)
ert_vcx64.tmf	vcx64	_rtwsfcn_vcx64	_int_ert_vcx64	_ert_vcx64	.lib
ert_unix.tmf	unix	_rtwsfcn	_int_ert	_ert	.a
grt_lcc64.tmf	lcc	n/a	n/a	_lcc	.lib
grt_vcx64.tmf	vcx64	n/a	n/a	_vcx64	.lib
grt_unix.tmf	unix	n/a	n/a	_std	.a

2 Set the precompiled library folder.

Use one of the following methods to set the precompiled library folder:

- Set the `TargetPreCompLibLocation` parameter, as described in "Specify the Location of Precompiled Libraries" on page 70-9.
- Set the `makeInfo.precompile` field in an `rtwmakecfg.m` function file. (For more information, see "Use rtwmakecfg.m API to Customize Generated Makefiles" on page 70-26.)

If you set `TargetPreCompLibLocation` and `makeInfo.precompile`, the setting for `TargetPreCompLibLocation` takes precedence.

The following command sets the precompiled library folder for model `my_model` to folder `lib` under the current working folder.

```
set_param(my_model,'TargetPreCompLibLocation', fullfile(pwd,'lib'));
```

Note: If you set both the target folder for the precompiled library files and a target library file suffix, the build process detects whether any precompiled library files are missing while processing builds.

3 Define a build specification.

Set up a structure that defines a build specification. The following table describes fields that you can define in the structure. These fields are optional, except for `rtwmakecfgDirs`.

Field	Description
`rtwmakecfgDirs`	A cell array of character vectors that name the folders containing `rtwmakecfg` files for libraries to be precompiled. The function uses the `Name` and `Location` elements of `makeInfo.library`, as returned by `rtwmakecfg`, to specify the name and location of the precompiled libraries. If you set the `TargetPreCompLibLocation` parameter to specify the library folder, that setting overrides the `makeInfo.library.Location` setting. **Note:** The specified model must contain blocks that use precompiled libraries specified by the `rtwmakecfg` files because the TMF-to-makefile conversion generates the library rules only if the build process uses the libraries.
`libSuffix`	A character vector that specifies the suffix, including the file type extension, to be appended to the name of each library (for example, `.a` or `_vc.lib`). The character vector must include a period (.). You must set the suffix with either this field or the `TargetLibSuffix` parameter. If you specify a suffix with both mechanisms, the `TargetLibSuffix` setting overrides the setting of this field.
`intOnlyBuild`	A Boolean flag. When set to true, the flag indicates the libraries are to be optimized such that they are compiled from integer code only. This field applies to ERT targets only.
`makeOpts`	A character vector that specifies an option to be included in the `rtwMake` command line.
`addLibs`	A cell array of structures that specify libraries to be built that are not specified by an `rtwmakecfg` function. Each structure must be defined with two fields that are character arrays: • `libName` — the name of the library without a suffix • `libLoc` — the location for the precompiled library The target makefile (TMF) can specify other libraries and how those libraries are built. Use this field to precompile those libraries.

The following commands set up build specification `build_spec`, which indicates that the files to be compiled are in folder `src` under the current working folder.

```
build_spec = [];
```

```
build_spec.rtwmakecfgDirs = {fullfile(pwd,'src')};
```

4 Issue a call to `rtw_precompile_libs`.

The call must specify the model for which you want to build the precompiled libraries and the build specification. For example:

```
rtw_precompile_libs(my_model,build_spec);
```

More About

- "Call Reusable External Algorithm Code for Simulation and Code Generation" on page 39-13
- "Place External C/C++ Code in Generated Code" on page 39-27
- "Call External Device Drivers" on page 39-38
- "Deploy Generated Standalone Executable Programs To Target Hardware" on page 49-2
- "Deploy Generated Component Software to Application Target Platforms" on page 49-22

Generate Component Source Code for Export to External Code Base

In this section...

If you have Embedded Coder software, you can generate function source code from modeling components to use in an external code base. The generated code does not include supporting scheduling code (for example, a step function). Controlling logic outside of the Simulink environment invokes the generated function code.

Modeling Options

You can generate function code to export for these modeling components:

- Export-function models (model containing functional blocks that consist exclusively of function-call subsystems, function-call model blocks, or other export-function models, as described in "Export-Function Models" (Simulink))
- Export-function subsystems (virtual subsystem that contains function-call subsystems)

To export code that the code generator produces for these modeling components, the modeling components must meet specific requirements on page 39-52.

For models designed in earlier releases, the code generator can export functions from triggered subsystems. The requirements stated for export-function subsystems also apply to exporting functions from triggered subsystems, with the following exceptions:

- Encapsulate triggered subsystems from which you intend to export functions in a top-level virtual subsystem.

- Triggered subsystems do not have to meet requirements and limitations identified for virtual subsystems that contain function-call subsystem.
- "Export Functions That Use Absolute or Elapsed Time" on page 39-53 does not apply to exporting functions from triggered subsystems.

Requirements

- Model solver must be a fixed-step discrete solver.
- You must configure each root-level Inport block that triggers a function-call subsystem to output a function-call trigger. These Inport blocks cannot connect to an Asynchronous Task Specification block.
- Model or subsystem, must contain only the following blocks at the root level:
  - Function-call blocks (such as Function-Call Subsystem, Simulink Function, S-Functions, and Function-Call Model blocks at the root level if the solver parameter **Tasking and sample time options > Periodic sample time constraint** is set to `Ensure sample time independent`)
  - Inport and Outport blocks (ports)
  - Constant blocks (including blocks that resolve to constants, such as Add)
  - Blocks with a sample time of Inf
  - Merge and data store memory blocks
  - Virtual connection blocks (such as, Function-Call Split, Mux, Demux, Bus Creator, Bus Selector, Signal Specification, and virtual subsystems that contain these blocks)
  - Signal-viewer blocks, such as Scope blocks (export-function subsystems only)
- When a constant block appears at the top level of the model or subsystem, you must set the model configuration parameter **Optimization > Signals and Parameters > Default parameter behavior** for the model or containing model to `Inlined`.
- Blocks inside the model or subsystem must support code generation.
- Blocks that use absolute or elapsed time must be inside a periodic function-call subsystem with a discrete sample time specified on the corresponding function-call root-level Inport block. See "Export Functions That Use Absolute or Elapsed Time" on page 39-53.
- Data signals that cross the boundary of an exported system cannot be a virtual bus and cannot be implemented as a Goto-From connection. Data signals that cross the export boundary must be scalar, muxed, or a nonvirtual bus.

In addition, for export-function models, data logging and signal-viewer blocks, such as the Scope block, are not allowed at the root level or within the function-call blocks.

For export-function subsystems, the following additional requirements apply:

- A trigger signal that crosses the boundary of an export-function subsystem must be scalar. Input and output data signals that do not act as triggers do not have to be scalar.
- When a constant signal drives an output port of an export-function subsystem, the signal must specify a storage class.

Export Functions That Use Absolute or Elapsed Time

If you want to export function code for a modeling component with blocks that use absolute or elapsed time, those blocks must be inside a function-call subsystem that:

- You configure for periodic execution
- You configure the root-level Inport block with a discrete sample time

To configure a function-call subsystem for periodic execution:

1. In the function-call subsystem, right-click the Trigger block and choose **Block Parameters** from the context menu.
2. In the **Sample time type** field, specify `periodic`.
3. Set the **Sample time** to the same granularity specified (directly or by inheritance) in the function-call initiator.
4. Click **OK** or **Apply**.

For more information, see "Absolute and Elapsed Time Computation" (Simulink Coder).

Limitations for Export-Function Subsystems

- Subsystem block parameters do not control the names of the files containing the generated code. The file names begin with the name of the exported subsystem.
- Subsystem block parameters 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) reflects the block from which the signal originates.
- The code generator cannot export reusable code for export-function subsystems. The **Code interface packaging** value `Reusable function` does not apply for export-function subsystems.

- You can export function-call systems for the C++ class code interface packaging only when its function specification is set to **Default step method**. See "Control Generation of C++ Class Interfaces" on page 26-23. The exported function is compatible with single-threaded execution. To avoid potential data race conditions for shared signals, invoke all members for the class from the same execution thread.
- The code generator supports a SIL or PIL block in accelerator mode only if its function-call initiator is noninlined in accelerator mode. Examples of noninlined initiators include Stateflow charts.
- A Level-2 S-function initiator block, such as a Stateflow chart or the built-in Function-Call Generator block, must drive a SIL block.
- You can export an asynchronous (sample-time) function-call system, but the software does not support the SIL or PIL block for an asynchronous system.
- The software does not support MAT-file logging for export-function subsystems. Specifications that enable MAT-file logging are ignored.
- The use of the TLC function `LibIsFirstInit` has been removed for export-function subsystems.

Workflow

To generate code for an exported function, iterate through the tasks listed in this table.

Task	Action	More Information
1	Review your assessment of external code characteristics and integration requirements.	"Choose an External Code Integration Workflow" on page 39-4
2	Verify that the model or subsystem that you are exporting satisfies function exporting requirements.	"Requirements" on page 39-52
3	Address data interface requirements by modifying the model or subsystem.	"Exchange Data Between External C/C++ Code and Simulink Model or Generated Code" on page 39-86
4	If necessary, configure function prototype.	"Configure Simulink Function Code Interface" on page 26-67 and, for fixed-step rate-based models, "Control Generation of Function Prototypes" on page 26-2 or "Control Generation of C++ Class Interfaces" on page 26-23

Task	Action	More Information
5	If necessary , update the model to place external application-specific code in generated system functions.	"Place External C/C++ Code in Generated Code" on page 39-27
6	Verify that the functions behave and perform as expected during simulation by creating and using a test harness model. The test harness model schedules execution of the functions during simulation.	"Configure Model, Generate Code, and Simulate" on page 33-2 and, if you have Simulink Test software, "Tests in Models" (Simulink Test)
7	Configure the model or subsystem for code generation.	"Generate Code Using Embedded Coder®", "Generate Code That Matches Appearance of External Code" on page 39-95, and"Model Configuration"
8	Generate code and a code generation report.	"Code Generation"
9	Review the generated code interface and static code metrics.	"Analyze the Generated Code Interface" on page 35-21 and "Static Code Metrics" on page 35-34
10	Build an executable program that includes the exported function code.	"Build Integrated Code Outside the Simulink Environment" on page 39-79
11	Verify that executable program behaves and performs as expected.	

Choose an Integration Approach

Multiple approaches are available for generating function code for export to an external development environment. The following table compares approaches. Choose the approach that aligns best with your integration requirements. For more information on how to create export-function models, see "Export-Function Models" (Simulink). For more information on generating code for function call subsystems, see "Generate Component Source Code for Export to External Code Base" on page 39-51.

Condition or Requirement	Use	More Information
• Traceability between modeling elements and generated code	Function-call subsystem	• "Generate C Function Code for Export-Function Model" on page 39-57

Condition or Requirement	Use	More Information
• Local inputs (Inport block) and outputs (Outport block)		• "Generate C++ Function and Class Code for Export-Function Model" on page 39-63 • "Function-Call Subsystems" (Simulink) • Function-Call Subsystem
• Control over generated function prototype • Formal input arguments (Argument Inport blocks) and output arguments (Argument Outport blocks) • Local inputs (Inport block) and outputs (Outport block)	Simulink Function block	• "Configure Simulink Function Code Interface" on page 26-67 • "Modeling Functions and Callers for Code Generation" on page 4-2 • "Generate Code for Functions and Callers" on page 4-6 • "Simulink Functions in Simulink Models" (Simulink) • Simulink Function
Code responds to an initialization event	Initialize Function block	• "Generate C Function Code for Export-Function Model" on page 39-57 • "Generate C++ Function and Class Code for Export-Function Model" on page 39-63 • "Generate Code That Responds to Initialize, Reset, and Terminate Events" on page 9-2 • "Create Model to Initialize, Reset, and Terminate State" (Simulink)

Condition or Requirement	Use	More Information
Code responds to a reset event	Reset Function block	• "Generate C Function Code for Export-Function Model" on page 39-57 • "Generate C++ Function and Class Code for Export-Function Model" on page 39-63 • "Generate Code That Responds to Initialize, Reset, and Terminate Events" on page 9-2 • "Create Model to Initialize, Reset, and Terminate State" (Simulink)
Code includes entry-point functions beyond what the code generator produces by default (*model*`_initialize`, *model*`_step`, and *model*`_terminate`)	S-function	"Write S-Function and TLC Files By Hand" on page 11-66
Single-model execution framework to use as test harness and to export code generated for portions of a model	Export-function subsystem	• "Code Generation of Subsystems" on page 3-2 • "Systems and Subsystems" (Simulink) • "Function-Call Subsystems" (Simulink) • Function-Call Subsystem

Generate C Function Code for Export-Function Model

This example shows how to generate function code for individual Simulink function blocks and function-call subsystems in a model without generating scheduling code.

To generate function code for export:

1. Create a model that contains the functions for export.
2. Create a test harness model that schedules execution of the functions during simulation.
3. Simulate the model that contains the functions by using the test harness model.
4. Generate code for the model that contains the functions.

Create Model That Contains Functions for Export

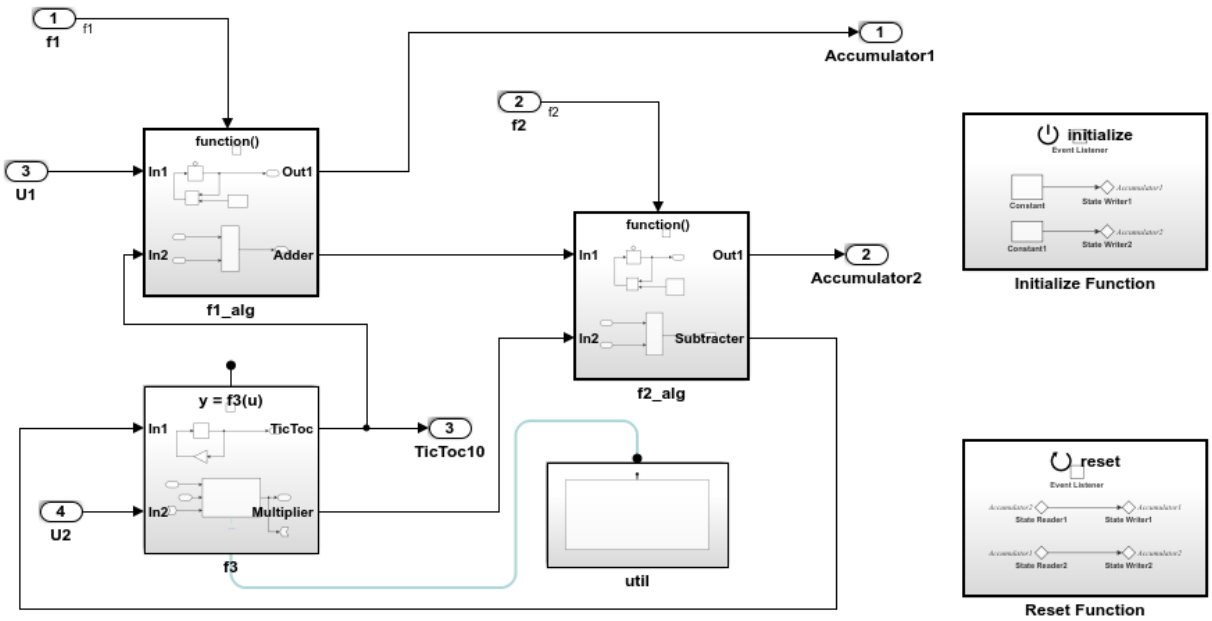
The model with functions for export must satisfy architectural constraints at the model root level. At the root level, valid blocks are:

- Inport
- Outport
- Function-Call Subsystem
- Simulink Function
- Goto
- From
- Merge

The code generator produces function code for Function-Call Subsystem and Simulink Function blocks and Initialize and Reset Function blocks. For a Function-call Subsystem block, you connect the block input ports to root Inport blocks that assert function-call signals. The subsystem is executed based on the function-call signal that it receives. A Simulink Function block is executed in response to the execution of a corresponding Function Caller block or Stateflow chart. An Initialize Function block is executed on a model initialize event and a Reset Function block is executed on a user-defined reset event.

For exporting functions, model `rtwdemo_functions` contains two function-call subsystems (`f1_alg` and `f2_alg`) and a Simulink Function block (`f3`) for exporting functions. The model also contains an Initialize Function block (`Initialize Function`) and a Reset Function block (`Reset Function`). To calculate initial conditions for blocks with state in other parts of the model, the State Writer blocks are used inside the Initialize and Reset Function blocks.

```
open_system('rtwdemo_functions')
```



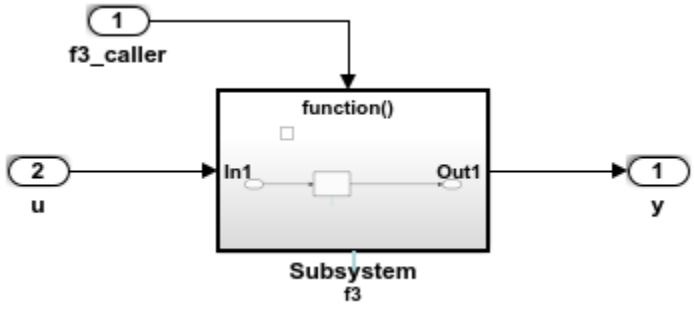
Create Model That Contains Function Caller Block

Use a Function Caller block to invoke a Simulink Function block. The Function Caller block can be in the same model or in a different model as the Simulink Function block.

Multiple Function Caller blocks can invoke a Simulink Function block. You can place the Function Caller block inside a function-call subsystem. During code generation, the code generator exports a function from the function-call subsystem.

The model `rtwdemo_caller` exports a function-call subsystem that contains a Function Caller block.

```
open_system('rtwdemo_caller')
```



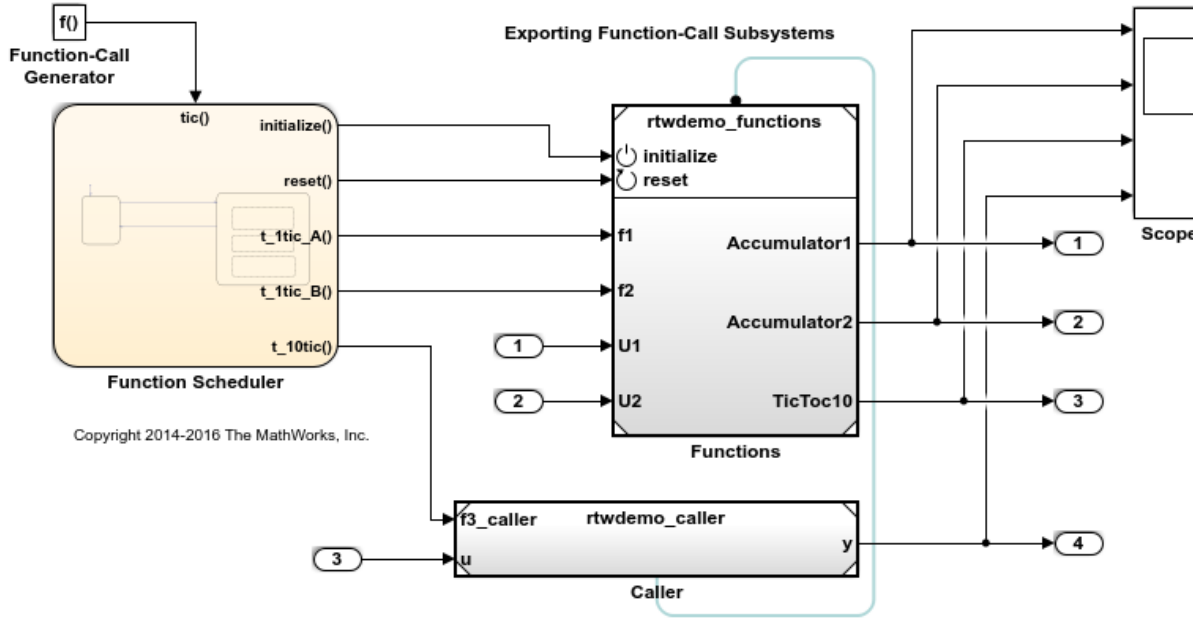
Create Test Harness Model for Simulation

When you export functions, the generated code does not include a scheduler. Create a test harness model to handle scheduling during simulation. Do not use the test harness model to generate code that you deploy.

Model `rtwdemo_export_functions` is a test harness. The model:

- Schedules the Simulink Function block with the Function Caller block in `rtwdemo_caller`.
- Provides function-call signals to other models in this example to schedule the model contents, including the model initialize and reset events.

```
open_system('rtwdemo_export_functions')
```



Simulate the Test Harness Model

Verify that the model containing the functions that you want to export is executed as you expect by simulating the test harness model. For example, simulate `rtwdemo_export_functions`.

```
sim('rtwdemo_export_functions')
```

Generate Function Code and Report

Generate code and a code generation report for the functions that you want to export. For example, generate code for `rtwdemo_functions`.

```
rtwbuild('rtwdemo_functions')
```

```
### Starting build procedure for model: rtwdemo_functions
### Successful completion of code generation for model: rtwdemo_functions
```

Review Generated Code

From the code generation report, review the generated code.

- `ert_main.c` is an example main program (execution framework) for the model. This code shows how to call the exported functions. The code also shows how to initialize and execute the generated code.
- `rtwdemo_functions.c` calls the initialization function, including `Initialize Function`, and exported functions for model components `f1_alg`, `f2_alg`, and `f3`.
- `rtwdemo_functions.h` declares model data structures and a public interface to the exported entry-point functions and data structures.
- `f3.h` is a shared file that declares the call interface for the Simulink function `f3`.
- `rtwtypes.h` defines data types, structures, and macros that the generated code requires.

Write Interface Code

Open and review the Code Interface Report. To write the interface code for your execution framework, use the information in that report.

1. Include the generated header files by adding directives `#include rtwdemo_functions.h`, `#include f3.h`, and `#include rtwtypes.h`.
2. Write input data to the generated code for model Inport blocks.
3. Call the generated entry-point functions.
4. Read data from the generated code for model Outport blocks.

Input ports:

- `rtU.U1` of type `real_T` with dimension 1
- `rtU.U2` of type `real_T` with dimension 1

Entry-point functions:

- Initialize entry-point function, `void rtwdemo_functions_initialize(void)`. At startup, call this function once.
- Reset entry-point function, `void rtwdemo_functions_reset(void)`. Call this function as needed.
- Exported function, `void f1(void)`. Call this function as needed.
- Exported function, `void f2(void)`. Call this function as needed.
- Simulink function, `void f3(real_T rtu_u, real_T *rty_y)`. Call this function as needed.

Output ports:

- `rtY.Accumulator1` of type `int8_T` with dimension [2]
- `rtY.Accumulator2` of type `int8_T` with dimension [2]
- `rtY.TicToc10` of type `int8_T` with dimension 1

More About

- "Generate Component Source Code for Export to External Code Base"
- "Deploy Generated Standalone Executable Programs To Target Hardware"
- "Customize Code Organization and Format"
- "Control Generation of Function Prototypes"
- "Modeling Functions and Callers for Code Generation" (Simulink Coder)
- "Generate Code for Functions and Callers" (Simulink Coder)

Close Example Models

```
bdclose('rtwdemo_export_functions')
bdclose('rtwdemo_functions')
bdclose('rtwdemo_caller')
```

Generate C++ Function and Class Code for Export-Function Model

This example shows how to generate function code for an export-function model that includes a function-call subsystem. The code generator produces function and class code that does not include scheduling code.

To generate function code for export:

1. Create a model that contains the functions for export.
2. Create a test harness model that schedules execution of the functions during simulation.
3. Simulate the model that contains the functions by using the test harness model.
4. Generate code for the model that contains the functions.

Create Model That Contains Functions and C++ Class Interface for Export

The model with functions for export with a C++ model class interface must satisfy architectural constraints at the model root level. For C++ class generation, blocks that are valid at the root level are:

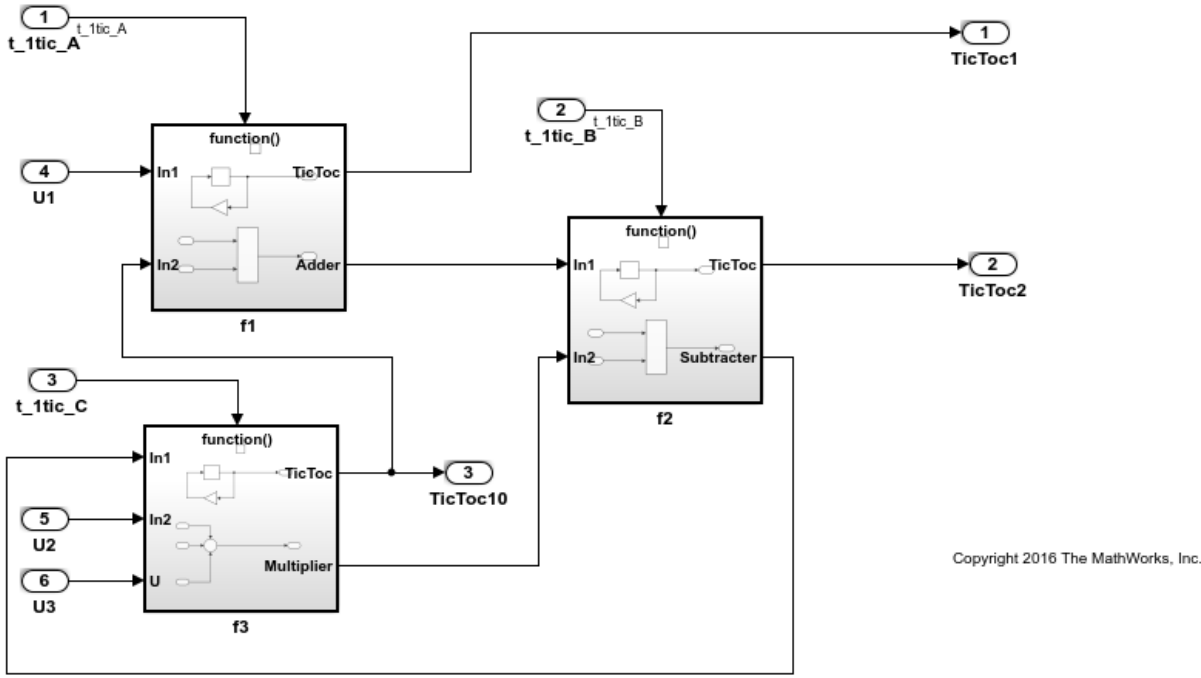
- Inport
- Outport
- Function-Call Subsystem
- Goto
- From
- Merge

Note: Export Function-Call Subsystem with C++ class interface does not support Simulink Function blocks.

The code generator produces function code for the Function-Call Subsystem block. For a Function-call Subsystem block, you connect the block input ports to root Inport blocks that assert function-call signals. The subsystem is executed based on the function-call signal that it receives.

Model `rtwdemo_cppclass_functions` contains function-call subsystems `f1`, `f2`, and `f3` for exporting functions.

```
open_system('rtwdemo_cppclass_functions')
```

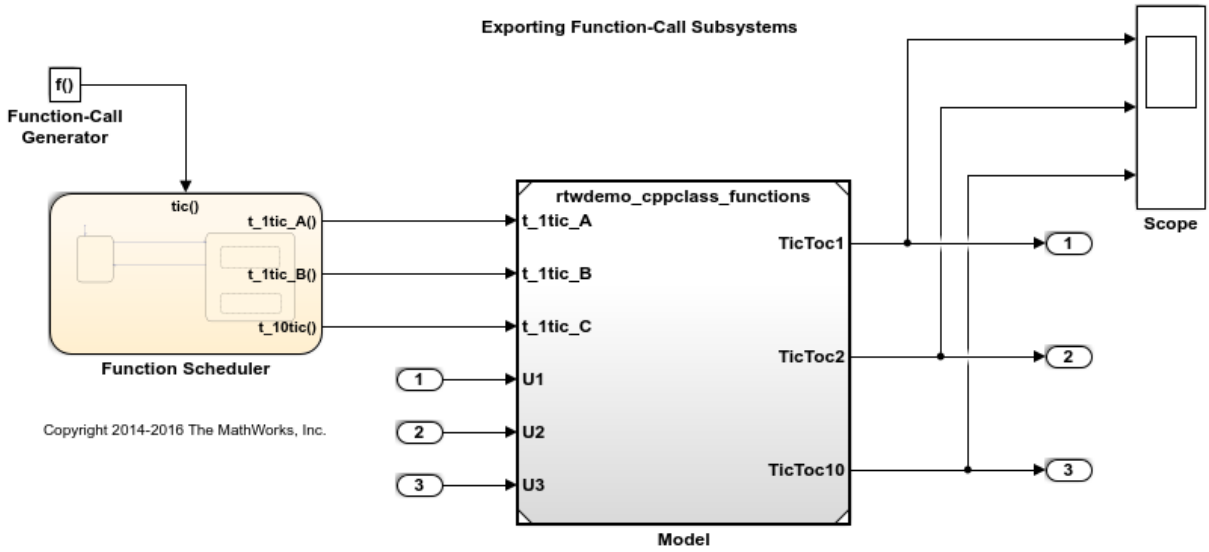


Create Test Harness Model for Simulation

When you export functions, the generated code does not include a scheduler. Create a test harness model to handle scheduling during simulation. Do not use the test harness model to generate code that you deploy.

Model `rtwdemo_cppclass_export_functions` is a test harness. The model provides function-call signals to other models in this example to schedule the model contents.

```
open_system('rtwdemo_cppclass_export_functions')
```



Simulate the Test Harness Model

Verify that the model containing the functions that you want to export is executed as you expect by simulating the test harness model. For example, simulate `rtwdemo_cppclass_export_functions`.

```
sim('rtwdemo_cppclass_export_functions')
```

Generate Function Code and Report

Generate code and a code generation report for the functions that you want to export. For example, generate code for `rtwdemo_cppclass_functions`.

```
rtwbuild('rtwdemo_cppclass_functions')

### Starting build procedure for model: rtwdemo_cppclass_functions
### Successful completion of build procedure for model: rtwdemo_cppclass_functions
```

Review Generated Code

From the code generation report, review the generated code.

- `ert_main.cpp` is an example main program (execution framework) for the model. This code shows how to call the exported functions. The code also shows how to initialize and execute the generated code.
- `rtwdemo_cppclass_functions.cpp` calls the initialization function, including `Initialize Function`, and exported functions for model subsystem components `f1`, `f2`, and `f3`.
- `rtwdemo_cppclass_functions.h` declares model data structures and a public interface to the exported entry-point functions and data structures.
- `rtwtypes.h` defines data types, structures, and macros that the generated code requires.

Write Interface Code

Open and review the Code Interface Report. To write the interface code for your execution framework, use the information in that report.

1. Include the generated header files by adding directives `#include rtwdemo_cppclass_functions.h` and `#include rtwtypes.h`.
2. Write input data to the generated code for model Inport blocks.
3. Call the generated entry-point functions.
4. Read data from the generated code for model Outport blocks.

Input ports:

- `rtU.U1` of type `real_T` with dimension 1
- `rtU.U2` of type `real_T` with dimension 1
- `rtU.U3` of type `real_T` with dimension 1

Entry-point functions:

- Initialize entry-point function, `void initialize(void)`. At startup, call this function once.
- Exported function, `void t_1tic_A(void)`. Call this function as needed.
- Exported function, `void t_1tic_B(void)`. Call this function as needed.
- Exported function, `void t_1tic_C(void)`. Call this function as needed.

Output ports:

- `rtY.TicToc1` of type `int8_T` with dimension [2]

- `rtY.TicToc2` of type `int8_T` with dimension [2]
- `rtY.TicToc10` of type `int8_T` with dimension 1

More About

- "Generate Component Source Code for Export to External Code Base"
- "Deploy Generated Standalone Executable Programs To Target Hardware"
- "Customize Code Organization and Format"
- "Control Generation of C++ Class Interfaces"
- "Modeling Functions and Callers for Code Generation" (Simulink Coder)
- "Generate Code for Functions and Callers" (Simulink Coder)

Close Example Models

```
bdclose('rtwdemo_cppclass_export_functions')
bdclose('rtwdemo_cppclass_functions')
```

Generate Code for Export-Function Subsystems

- "Specify a Custom Initialize Function Name" on page 39-69
- "Specify a Custom Description" on page 39-69
- "Optimize Code Generated for Export-Function Subsystems" on page 39-70

To generate code for an export-function subsystem:

1. Verify that the subsystem for which you are generating code satisfies exporting requirements on page 39-52.
2. In the Configuration Parameters dialog box:
   a. On the **Code Generation** pane, specify an ERT-based system target file, such as `ert.tlc`.
   b. If you want a SIL block with the generated code, for verification purposes, from the **Configuration Parameters** > **All Parameters** > **Create block** drop-down list, select `SIL`.
   c. Click **OK** or **Apply**.
3. Right-click the subsystem block and choose **C/C++ Code** > **Export Functions** from the context menu.

The `Build code for subsystem: Subsystem` dialog box is not specific to export-function subsystems. Generating code does not require entering information in the dialog box.

4 Click **Build**.

The code generator produces code and places it in the working folder.

If you set **Create block** to `SIL` 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 do for generated ERT code and S-function block. For optional workflow tasks, see "Specify a Custom Initialize Function Name" on page 39-69 and "Specify a Custom Description" on page 39-69.

Specify a Custom Initialize Function Name

You can specify a custom name for the initialize function of your exported function as an argument to the `rtwbuild` command. The command takes the following form:

```
blockHandle = rtwbuild('subsystem', 'Mode', 'ExportFunctionCalls',..
            'ExportFunctionInitializeFunctionName', 'fcnname')
```

`fcnname` specifies the function name. For example, if you specify the name `'myinitfcn'`, the build process emits code similar to:

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

Specify a Custom Description

You can enter a custom description for an exported function by using the Block Properties dialog box of an Inport block.

1 Right-click the Inport block that drives the control port of the subsystem for which you are exporting code.

2 Select **Properties**.

3 In the **General** tab, in the **Description** field, enter your descriptive text.

During function export, the text you enter is emitted to the generated code in the header for the Inport block. For example, if you open the example program `rtwdemo_exporting_functions` and enter a description in the Block Properties dialog box for port `t_1tic_A`, the code generator produces code that is similar to:

```
/*
 * Output and update for exported function: t_1tic_A
 *
 *  My custom description of the exported function
*/
void t_1tic_A(void)
{
...
}
```

Optimize Code Generated for Export-Function Subsystems

To optimize the code generated for an export-function subsystem, specify a separate storage class for each input signal and output signal that crosses the boundary of the subsystem.

For each function-call subsystem that you are exporting:

1. Right-click the subsystem.
2. From the context menu, choose **Block Parameters (Subsystem)**.
3. Select the **Code Generation** tab.
4. Set **Function packaging** to `Auto`.
5. Click **OK** or **Apply**.

More About

- "Design Models for Generated Embedded Code Deployment" on page 1-2
- "Systems and Subsystems" (Simulink)
- "Triggered Subsystems" (Simulink)
- "Function-Call Subsystems" (Simulink)
- "Export-Function Models" (Simulink)

Generate Shared Library for Export to External Code Base

In this section...

About Generated Shared Libraries

If you have Embedded Coder software, you can generate a shared library—Windows dynamic link library (`.dll`), UNIX shared object (`.so`), or Macintosh OS X dynamic library (`.dylib`)— from a model component. You or others can integrate the shared library into an application that runs on a Windows, UNIX, or Macintosh OS X development computer. Uses of shared libraries include:

- Adding a software component to an application for system simulation
- Reusing software modules among applications on a development computer
- Hiding intellectual property associated with software that you share with vendors

When producing a shared library, the code generator exports:

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

Workflow

To generate a shared library from a model component and use the library, complete the tasks listed in this table.

Task	Action	More Information
1	Review your assessment of external code characteristics and integration requirements.	• "Choose an External Code Integration Workflow" on page 39-4 • "Shared Library Limitations" on page 47-7
2	Configure the model for code generation.	"Generate Code That Matches Appearance of External Code" on page 39-95 and "Model Configuration"
3	Configure the model for the code generator to produce a shared library and initiate code generation.	"Generate Shared Libraries" on page 39-73
4	Verify that the generated shared library meets requirements. For example, review the code generation report and view the list of symbols in the library. • On Windows, use the Dependency Walker utility, downloadable from www.dependencywalker.com • On UNIX, use `nm -D model.so` • On Macintosh OS X, use `nm -g model.dylib`	
5	Use the shared library in application code.	"Create Application Code That Uses Generated Shared Libraries" on page 39-73
6	Compile and link application code that loads and uses the generated shared library.	"Build Integrated Code Outside the Simulink Environment" on page 39-79
7	Verify that executable program behaves and performs as expected.	

Generate Shared Libraries

1. When configuring the model for code generation, select the system target file `ert_shrlib.tlc`.
2. Build the model. The code generator produces source code for the model and a shared library version of the code. The code generator places the source code in the code generation folder and the shared library (`.dll`, `.so`, or `.dylib` file) in your current working folder. The code generator also produces and retains a `.lib` file to support implicit linking.

Create Application Code That Uses Generated Shared Libraries

This example application code is generated for the example "Interface to a Development Computer Simulator By Using a Shared Library" on page 39-76.

1. Create an application header file that contains type declarations for model external input and output. For example:

```
#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_*/
```

2. In the application C source code, dynamically load the shared library. Use preprocessing conditional statements to invoke platform-specific commands. For example:

```
#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));
}
```

3 From the application C source code, access exported data and functions generated from the model. The code uses hooks to add 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);
    }
```

Limitations

- Code generation for the `ert_shrlib.tlc` system target file exports the following as data:
  - Variables and signals of type `ExportedGlobal`
  - Real-time model structure (*`model`*_M)
- Code generation for the `ert_shrlib.tlc` system target file supports the C language only (not C++). When you select `ert_shrlib.tlc`, language selection is unavailable on the **Code Generation** pane in the Configuration Parameters dialog box.
- To reconstruct a model simulation by using a generated shared library, the application author must maintain the timing between system and shared library function calls in the original application. The timing must be consistent so that you can compare the simulation and integration results. Additional simulation considerations apply if generating a shared library from a model that enables model configuration parameters **Support: continuous time** and **Single output/update function**. For more information, see Single output/update function (Simulink Coder) dependencies.

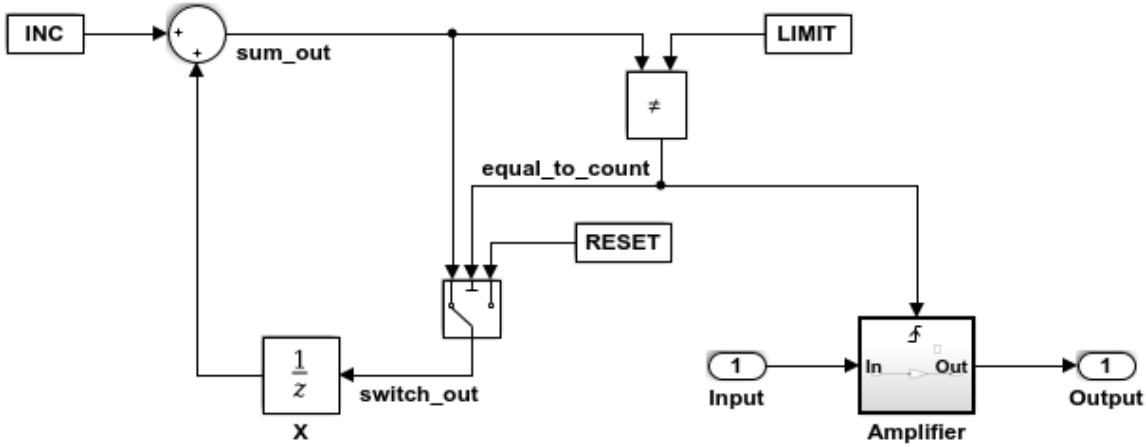
Interface to a Development Computer Simulator By Using a Shared Library

This example generates a shared library for interfacing to a simulator that runs on your development computer. Generate the shared library by using the system target file `ert_shrlib.tlc`.

To build a shared library from the model and use the library in an appplication:

1. Develop your model. For this example, open the model `rtwdemo_shrlib`.

```
open_system('rtwdemo_shrlib');
```



Description
This example shows the use of a shared library generated using the system target file ert_shrlib.tlc for a single-rate discrete-time model. An 8-bit counter feeding a triggered subsystem is parameterized with constants INC=1, LIMIT=4, and RESET=0. The I/O for the model is Input and Output. The Amplifier subsystem amplifies the input signal by a gain factor K=3. The output signal is updated whenever signal equal_to_count is true. The shared library generated from this example can be dynamically loaded from another application. See rtwdemo_shrlib_app.c for an application example written in C language.

Instructions
1. View the Code Generation configuration by clicking the yellow button below.
2. Double-click the blue button to build and run an example application that uses the generated shared library. Click the white buttons to view source code.

matlabroot\toolbox\rtw\rtwdemos\shrlib_demo\rtwdemo_shrlib_app.h

matlabroot\toolbox\rtw\rtwdemos\shrlib_demo\rtwdemo_shrlib_app.c

matlabroot\toolbox\rtw\rtwdemos\shrlib_demo\run_rtwdemo_shrlib_app.m

Run script run_rtwdemo_shrlib_app.m (double-click)

View Code Generation Configuration (double-click)

The model is a single-rate, discrete-time model. An 8-bit counter feeds the triggered subsystem named `Amplifier`. Parameters `INC`, `LIMIT`, and `RESET` are set to constant

values 1, 4, and 0, respectively. When signal `equal_to_count` is true, the subsystem amplifies its input signal by a gain factor `K=3` and the output signal is updated.

2. Configure the model for code generation, specifying `ert_shrlib.tlc` as the system target file. Click the yellow button on the model to view the configuration settings on the **Code Generation** pane in the Configuration Parameters dialog box.

3. Build the shared library file. The file that the code generator produces depends on your development platform. For example, on a Windows system, the code generator produces the library file `rtwdemo_shrlib_win64.dll`.

4. Create application code that uses the shared library. This example uses application code that is available in the these files:

```
matlabroot\toolbox\rtw\rtwdemos\shrlib_demo\rtwdemo_shrlib_app.h
matlabroot\toolbox\rtw\rtwdemos\shrlib_demo\rtwdemo_shrlib_app.c
```

To view the source code in these files, in the model, click the white buttons for the `.h` and `.c` files.

5. Compile and link the file application and shared library files to produce an executable program. The following script compiles, builds, and runs the program.

```
matlabroot\toolbox\rtw\rtwdemos\shrlib_demo\rtwdemo_shrlib_app.m
```

To view the script code, in the model, click the white button for the `.m` file.

To build the model and run the application that uses the generated shared library, in the model, double-click the blue button.

For more information about using a shared library, see “Package Generated Code as Shared Libraries”.

More About

- “Design Models for Generated Embedded Code Deployment” on page 1-2
- “Select a System Target File” on page 30-2
- “Manage Build Process Files” on page 33-42
- “Model Protection”

Build Integrated Code Outside the Simulink Environment

Identify required files and interfaces for calling generated code in an external build process.

Learn how to:

- Collect files required for building integrated code outside of Simulink®.
- Interface with external variables and functions.

For information about the example model and related examples, see "Generate C Code from a Control Algorithm for an Embedded System".

Collect and Build Required Data and Files

The code that Embedded Coder® generates requires support files that MathWorks® provides. To relocate the generated code to another development environment, such as a dedicated build system, you must relocate these support files. You can package these files in a zip file by using the `packNGo` utility. This utility finds and packages the files that you need to build an executable image. The utility uses tools for customizing the build process after code generation, which include a `buildinfo_data` structure, and a `packNGo` function. These files include external files that you identify in the **Code Generation > Custom Code** pane in the Model Configuration Parameters dialog box. The utility saves the `buildinfo` MAT-file in the *`model`*`_ert_rtw` folder.

Open the example model, rtwdemo_PCG_Eval_P5.

This model is configured to run `packNGo` after code generation.

Generate code from the entire model.

To generate the zip file manually:

1. Load the file `buildInfo.mat` (located in the `rtwdemo_PCG_Eval_P5_ert_rtw` subfolder).
2. At the command prompt, enter the command `packNGo(buildInfo)`.

The number of files in the zip file depends on the version of Embedded Coder® and on the configuration of the model that you use. The compiler does not require all of the files in the zip file. The compiled executable size (RAM/ROM) depends on the linking process. The linker likely includes only the object files that are necessary.

Integrating the Generated Code into an Existing System

This example shows how to integrate the generated code into an existing code base. The example uses the Eclipse™ IDE and the Cygwin™/gcc compiler. The required integration tasks are common to all integration environments.

Overview of Integration Environment

A full embedded controls system consists of multiple hardware and software components. Control algorithms are just one type of component. Other components can be:

- An operating system (OS)
- A scheduling layer
- Physical hardware I/O
- Low-level hardware device drivers

Typically, you do not use the generated code in these components. Instead, the generated code includes interfaces that connect with these components. MathWorks® provides hardware interface block libraries for many common embedded controllers. For examples, see the Embedded Targets block library.

This example provides files to show how you can build a full system. The main file is `example_main.c`, which contains a simple main function that performs only basic actions to exercise the code.

View example_main.c.

```
int_T main(void)
{
  /* Initialize model */
  rt_Pos_Command_Arbitration_Init(); /* Set up the data structures for chart*/
  rtwdemo_PCG__Define_Throt_Param(); /* SubSystem: '<Root>/Define_Throt_Param' */
  defineImportData();                /* Defines the memory and values of inputs */

  do /* This is the "Schedule" loop.
      Functions would be called based on a scheduling algorithm */
  {
    /* HARDWARE I/O */

    /* Call control algorithms */
    PI_Cntrl_Reusable((*pos_rqst),fbk_1,&rtwdemo_PCG_Eval_P5_B.PI_ctrl_1,
                      &rtwdemo_PCG_Eval_P5_DWork.PI_ctrl_1);
    PI_Cntrl_Reusable((*pos_rqst),fbk_2,&rtwdemo_PCG_Eval_P5_B.PI_ctrl_2,
                      &rtwdemo_PCG_Eval_P5_DWork.PI_ctrl_2);
    pos_cmd_one = rtwdemo_PCG_Eval_P5_B.PI_ctrl_1.Saturation1;
    pos_cmd_two = rtwdemo_PCG_Eval_P5_B.PI_ctrl_2.Saturation1;

    rtwdemo_Pos_Command_Arbitration(pos_cmd_one, &Throt_Param, pos_cmd_two,
                                    &rtwdemo_PCG_Eval_P5_B.sf_Pos_Command_Arbitration);
```

The file:

- Defines function interfaces (function prototypes).
- Includes files that declare external data.
- Defines `extern` data.
- Initializes data.
- Calls simulated hardware.
- Calls algorithmic functions.

The order of function execution matches the order of subsystem execution in the test harness model and in `rtwdemo_PCG_Eval_P5.h`. If you change the order of execution in `example_main.c`, results that the executable image produces differ from simulation results.

Match System Interfaces

Integration requires matching the *Data* and *Function* interfaces of the generated code and the existing system code. In this example, the `example_main.c` file imports and exports the data through `#include` statements and `extern` declarations. The file also calls the functions from the generated code.

Connect Input Data

The system has three input signals: `pos_rqst`, `fbk_1`, and `fbk_2`. The generated code accesses the two feedback signals through direct reference to imported global variables (storage class `ImportedExtern`). The code accesses the position signal through an imported pointer (storage class `ImportedExternPointer`).

The handwritten file `defineImportedData.c` defines the variables and the pointer. The generated code does not define the variables and the pointer because the handwritten code defines them. Instead, the generated code declares the imported data (`extern`) in the file `rtwdemo_PCG_Eval_P5_Private.h`. In a real system, the data typically comes from other software components or from hardware devices.

View defineImportedData.c.

```
/* Define imported data */
#include "rtwtypes.h"
#include "defineImportedData.h"

real_T fbk_1;
real_T fbk_2;
real_T dummy_pos_value = 10.0;
real_T *pos_rqst;
void defineImportData(void)
{
    pos_rqst = &dummy_pos_value;
}
```

View rtwdemo_PCG_Eval_P5_Private.h.

```
/* Imported (extern) block signals */
extern real_T fbk_1;                   /* '<Root>/fbk_1' */
extern real_T fbk_2;                   /* '<Root>/fbk_2' */

/* Imported (extern) pointer block signals */
extern real_T *pos_rqst;               /* '<Root>/pos_rqst' */
```

Connect Output Data

In this example, you do not access the output data of the system. The example "Test Generated Code" shows how you can save the output data to a standard log file. You can access the output data by referring to the file `rtwdemo_PCG_Eval_P5.h`.

View rtwdemo_PCG_Eval_P5.h.

Access Additional Data

The generated code contains several structures that store commonly used data including:

- Block state values (integrator, transfer functions)
- Local parameters
- Time

The table lists the common data structures. Depending on the configuration of the model, some or all of these structures appear in the generated code. The data is declared in the file `rtwdemo_PCG_Eval_P5.h`, but in this example, you do not access this data.

```
Data Type      Data Name     Data Purpose

Constants      |model_cP|    Constant parameters
Constants      |model_cB|    Constant block I/O
Output         |model_U|     Root and atomic subsystem input
Output         |model_Y|     Root and atomic subsystem output
Internal data  |model_B|     Value of block output
Internal data  |model_D|     State information vectors
Internal data  |model_M|     Time and other system level data
Internal data  |model_Zero|  Zero-crossings
Parameters     |model_P|     Parameters
```

Match Function Call Interfaces

By default, functions that the code generator generates have a `void` *`Func`*`(void)` interface. If you configure the model or atomic subsystem to generate reentrant code,

the code generator creates a more complex function prototype. In this example, the `example_main` function calls the generated functions with the correct input arguments.

```
PI_Cntrl_Reusable((*pos_rqst),fbk_1,&rtwdemo_PCG_Eval_P5_B.PI_ctrl_1,
                  &rtwdemo_PCG_Eval_P5_DWork.PI_ctrl_1);
PI_Cntrl_Reusable((*pos_rqst),fbk_2,&rtwdemo_PCG_Eval_P5_B.PI_ctrl_2,
                  &rtwdemo_PCG_Eval_P5_DWork.PI_ctrl_2);
pos_cmd_one = rtwdemo_PCG_Eval_P5_B.PI_ctrl_1.Saturation1;
pos_cmd_two = rtwdemo_PCG_Eval_P5_B.PI_ctrl_2.Saturation1;

rtwdemo_Pos_Command_Arbitration(pos_cmd_one, &Throt_Param, pos_cmd_two,
                                &rtwdemo_PCG_Eval_P5_B.sf_Pos_Command_Arbitration);
```

Calls to the function `PI_Cntrl_Reusable` use a mixture of separate, unstructured global variables and Simulink® Coder™ data structures. The handwritten code defines these variables. The structure types are defined in `rtwdemo_PCG_Eval_P5.h`.

Build Project in Eclipse™ Environment

This example uses the Eclipse™ IDE and the Cygwin™ GCC debugger to build the embedded system. The example provides installation files for both programs. Software components and versions numbers are:

- Eclipse™ SDK 3.2
- Eclipse™ CDT 3.3
- Cygwin™/GCC 3.4.4-1
- Cygwin™/GDB 20060706-2

To install and use Eclipse™ and GCC, see “Install and Use Cygwin and Eclipse”.

You can install the files for this example by clicking this hyperlink:

Set up the build folder.

Alternatively, to install the files manually:

1. Create a build folder (`Eclipse_Build_P5`).
2. Unzip the file `rtwdemo_PCG_Eval_P5.zip` into the build folder.
3. Delete the files `rtwdemo_PCG_Eval_P5.c`, `ert_main.c` and `rt_logging.c`, which are replaced by `example_main.c`.

You can use the Eclipse™ debugger to step through and evaluate the execution behavior of the generated C code. See the example “Install and Use Cygwin and Eclipse”.

To exercise the model with input data, see “Test Generated Code”.

Related Topics

- “Generate Component Source Code for Export to External Code Base”
- “Generate Shared Library for Export to External Code Base”

Exchange Data Between External C/C++ Code and Simulink Model or Generated Code

Whether you import your external code into a Simulink model (for example, by using the Legacy Code Tool) or export the generated code to an external environment, the model or the generated code can exchange data (signals, states, and parameters) with your code.

Functions in C or C++ code, including your function, can exchange data with a caller or a called function through:

- Arguments (formal parameters) of functions. When a function exchanges data through arguments, a caller can call the function multiple times. Each instance of the called function can manipulate its own independent set of data so that the instances do not interfere with each other.
- Direct access to global variables. Global variables can:
  - Enable different algorithms (functions) and instances of the same algorithm to share data such as calibration parameters and error status.
  - Enable the different rates (functions) of a multitasking system to exchange data.
  - Enable different algorithms to exchange data asynchronously.

In Simulink, you can organize and configure data so that a model uses these exchange mechanisms to provide, extract, and share data with your code.

Before you attempt to match data interfaces, to choose an integration approach, see “Choose an External Code Integration Workflow” on page 39-4.

In this section...

Import External Code into Model

To exchange data between your model and your external function, choose an exchange mechanism based on the technique that you chose to integrate the function.

- To exchange data through the arguments of your external function, construct and configure your model to create and package the data according to the data types of the arguments. Then, you connect and configure the block that calls or represents your function to accept, produce, or refer to the data from the model.

  For example, if you use the Legacy Code Tool to generate an S-Function block that calls your function, the ports and parameters of the block correspond to the arguments of the function. You connect the output signals of upstream blocks to the input ports and set parameter values in the block mask. Then, you can create signal lines from the output ports of the block and connect those signals to downstream blocks.
- To exchange data through global variables that your external code already defines, a best practice is to use a Stateflow chart to call your function and to access the variables. You write algorithmic C code in the chart so that during simulation or execution of the generated code the model reads and writes to the variables.

  To use such a global variable as an item of parameter data (not signal or state data) elsewhere in a model, you can create a numeric MATLAB variable or `Simulink.Parameter` object that represents the variable. If you change the value of the C-code variable in between simulation runs, you must manually synchronize the value of the Simulink variable or object. If your algorithmic code (function) changes the value of the C-code variable during simulation, the corresponding Simulink variable or object does not change.

  If you choose to create a Simulink representation of the C-code variable, you can configure the Simulink representation so that the generated code reads and writes to the variable but does not duplicate the variable definition. Apply a storage class to the Simulink representation.

Technique for Integrating External Function	Mechanism to Exchange Data with Model	Examples and More Information
S-Function block	Function arguments	To call your function through an S-function that you create by using the Legacy Code Tool, see “Integrate C Functions into Simulink Models with Legacy Code Tool” (Simulink).
Stateflow chart	Function arguments and direct access to global variables	To call your function and access global variables in a Stateflow chart, see “Integrate Custom C/C++ Code for Simulation” (Stateflow). For information about

Technique for Integrating External Function	Mechanism to Exchange Data with Model	Examples and More Information
		creating data items in a chart (which you can pass to your function as arguments), see "Add Stateflow Data" (Stateflow).
`coder.ceval` in MATLAB Function block	Function arguments	To call your function in a MATLAB Function block by using `coder.ceval`, see "Integrate C Code Using the MATLAB Function Block" (Simulink). For information about creating data items in a MATLAB Function block (which you can pass to your function as arguments), see "Ports and Data Manager" (Simulink).

Export Generated Code to External Environment

To export the generated code into your external code so that you can later compile and deploy the code, you configure the generated code to match the data interface of your external code. For example, if your external code defines some global variables for storing input data and expects the generated code to calculate that input data, you can construct the model and configure Outport blocks so that the generated code interacts with the variables.

- You can generate reentrant code from a model, which means that the generated entry-point functions exchange data through arguments. If you have Embedded Coder, you can control the prototypes of the entry-point functions. For example, you can make root-level Inport blocks appear in the generated code as arguments of the entry-point functions, which enables your external code to call the functions multiple times. For more information about generating reentrant code from a model, see "Generate Reentrant Code from Top-Level Models" on page 25-4. For more information about function prototype control, see "Control Generation of Function Prototypes" on page 26-2.

  Similarly, when you generate code from a subsystem, you can generate reentrant code and configure the function prototypes by using parameters of the Subsystem block. For more information, see "Generate Reusable Function for Identical Subsystems Within a Model" on page 3-11 and `RTW.configSubsystemBuild`.
- You can exchange data between the generated code and your code through global variables. You can generate variable definitions for your code to use, or you can share and reuse existing variables that your code already defines.

To make the generated code read or write to an item of signal, state, or parameter data as a global variable, apply a storage class or custom storage class to the data. The storage class also determines whether the generated code exports the variable definition (memory allocation) to your external code or imports the definition from your code. For information about controlling a data interface by applying storage classes in a model, see "Design Data Interface by Configuring Inport and Outport Blocks" on page 19-134 and "Configure Data Interface by Applying Custom Storage Classes".

Simulink Representations of C Data Types and Constructs

To model and reuse your custom C data types such as structures, enumerations, and `typedef` aliases, use the information in these tables.

Modeling Patterns for Matching C-Code Data

C Data Type or Construct	Example External Code	Simulink Equivalent	More Information
Primitive type alias (`typedef`)	`typedef float mySinglePrec_T`	Create a `Simulink.AliasType` object. Use the object to: • Set the data types of signals and block parameters in a model. • Configure data type replacements for code generation. Generating code that uses an alias requires Embedded Coder.	For information about defining custom data types for your model, see `Simulink.AliasType` and "What Are User-Defined Data Types?" on page 21-2. For an example that shows how to export the generated code into your external code, see "Conform to Coding Standards by Replacing and Renaming Data Types" on page 21-22.
Array	`int myArray[6];`	Specify signal and parameter dimensions	For information about how the generated code

C Data Type or Construct	Example External Code	Simulink Equivalent	More Information
		as described in "Signal Dimensions" (Simulink).	stores nonscalar data (including limitations), see "Code Generation of Matrices and Arrays" (Simulink Coder). For an example that shows how to export the generated code into your external code, see "Reuse Parameter Data from Custom Code in the Generated Code" on page 23-17. To model lookup tables, see Simulink.LookupTable.

Enumeration	`typedef enum myColorsType {`
	`  Red = 0,`
	`  Yellow,`
	`  Blue`
	`} myColorsType;`
		Define a Simulink enumeration that corresponds to your enumeration definition. Use the Simulink enumeration to set data types in a model.	To use enumerated data in a Simulink model, see "Use Enumerated Data in Simulink Models" (Simulink).
			
			For an example that shows how to export the generated code into your external code, see "Exchange Structured and Enumerated Data Between Generated and External Code" on page 21-28.

C Data Type or Construct	Example External Code	Simulink Equivalent	More Information

Structure	`typedef struct myStructType {`
	`  int count;`
	`  double coeff;`
	`} myStructType;`
		Create a `Simulink.Bus` object that corresponds to your structure type.
		
		To create structured signal or state data, package multiple signal lines in a model into a single nonvirtual bus signal.
		
		To create structured parameter data, create a parameter object (such as `Simulink.Parameter`) that stores a MATLAB structure. Use the bus object as the data type of the parameter object.
		
		To package lookup table data into a structure, use `Simulink.LookupTable` and, optionally, `Simulink.Breakpoint` objects.
			For information about bus signals, see "Getting Started with Buses" (Simulink). For information about structures of parameters, see "Organize Related Block Parameter Definitions in Structures" (Simulink).
			
			For an example that shows how to import your external code into a model by using the Legacy Code Tool, see "Integrate C Function Whose Arguments Are Pointers to Structures" (Simulink).
			
			For examples that show how to export the generated code into your external code, see "Exchange Structured and Enumerated Data Between Generated and External Code" on page 21-28 and "Access Structured Data Through

C Data Type or Construct	Example External Code	Simulink Equivalent	More Information
			a Pointer That External Code Defines" on page 23-27. To package lookup table data into a structure, Simulink.LookupTable.

Additional Modeling Patterns for Code Generation

C Data Type or Construct	Example External C Code	Simulink Equivalent	More Information

Bit field	`typedef struct myBitField {`
	`  unsigned short int MODE : 1;`
	`  unsigned short int FAIL : 1;`
	`  unsigned short int OK : 1;`
	`} myBitField`
		Apply the custom storage class `BitField` to signals, states, and parameters whose data type is `boolean`.
		
		Use a model configuration parameter to aggregate Boolean data into bit fields.
		
		This technique requires Embedded Coder.
			"Bitfields" on page 13-95
			
			"Optimize Generated Code By Packing Boolean Data Into Bitfields" on page 57-14

Macro	`#define myParam 9.8`	Apply the custom storage classes `Define` and `ImportedDefine` to parameters.	"Macro Definitions (#define)" on page 13-77

C Data Type or Construct	Example External C Code	Simulink Equivalent	More Information
		This technique requires Embedded Coder.	
Storage type qualifiers such as `const` and `volatile`	`const volatile int countLimit;`	Apply the custom storage classes `Const`, `Volatile`, or `ConstVolatile` to signals (`volatile`), states, and parameters (`const`, `volatile`, or `const volatile`). This technique requires Embedded Coder.	"Type Qualifiers" on page 13-15

Function call that reads or writes to data	`/* Call this function to acquire the value of the signal. */`
	`double get_inSig(void)`
	`{`
	`  return myBigStruct.inSig;`
	`}`
		Apply the custom storage class `GetSet` to signals, states, and parameters. Each data item appears in the generated code as a call to your custom functions that read and write to the target data.
		
		This technique requires Embedded Coder.
			"Access Data Through Functions with Custom Storage Class `GetSet`" on page 23-92

Related Examples

- "Generate Code That Matches Appearance of External Code" on page 39-95
- "Exchange and Reuse Parameter Data Between Generated Code and Existing Code" on page 23-11

- "Design Data Interface by Configuring Inport and Outport Blocks" on page 19-134
- "Configure Generated Code According to Interface Control Document" on page 23-112
- "Generate Code That Dereferences Data from a Literal Memory Address" on page 23-83

Generate Code That Matches Appearance of External Code

A key aspect of code integration, especially for larger projects, is adherence to guidelines and standards for code appearance. If code appearance requirements apply to your project, review the requirements in this table. To learn more, see the relevant information.

Requirement	More Information
Thoroughly document code or document code in a specific way.	• "Configure Code Comments" on page 28-14 • "Specify Comment Style" on page 36-14 • "Add Custom Comments to Generated Code" on page 36-3 • "Add Custom Comments for Variables in the Generated Code" on page 36-5 • "Add Global Comments" on page 36-8 • "Annotate Code for Justifying Polyspace Checks" on page 36-98
Control the length and naming of code identifiers (symbols), including the use of reserved names.	• "Identifier Format Control" on page 36-22 • "Specify Identifier Length to Avoid Naming Collisions" on page 28-17 • "Specify Reserved Names for Generated Identifiers" on page 28-18 • "Customize Generated Identifier Naming Rules" on page 36-15 • "Avoid Identifier Name Collisions with Referenced Models" on page 36-30 • "Control Name Mangling in Generated Identifiers" on page 36-28 • "Specify Boolean and Data Type Limit Identifiers" on page 21-43
Control style aspects of code: • Level of parenthesization • Order of operations in expressions	"Control Code Style" on page 36-36

Requirement	More Information
• Empty primary condition expressions `in` if statements • `if-elseif-else` or `switch-case` statements for decision logic • Use of `extern` keyword in function declarations • Use of `default` cases for `switch-case` statements • Use multiplications by powers of two or signed bitwise shifts • Cast expressions • Indent style (Kernighan and Ritchie or Allman) and size	
Placement of data definitions and declarations, including location of global identifiers and global data declarations (`extern`)	"Manage Placement of Data Definitions and Declarations" on page 36-100
Appearance of code for flow charts	"Enhance Readability of Code for Flow Charts" on page 36-127
Apply code templates to control organization of code and use of banners	• "Customize Code Organization and Format" on page 36-54 • "Template Symbols and Rules" on page 36-90 • "Specify Templates For Code Generation" on page 36-56 • "Generate Custom File and Function Banners" on page 36-82 • "Change the Organization of a Generated File" on page 36-65 • "Generate Source and Header Files with a Custom File Processing (CFP) Template" on page 36-67

More About

- "Code Appearance"

- “Choose an External Code Integration Workflow” on page 39-4

40

Program Building, Interaction, and Debugging in Simulink Coder

Select C or C++ Programming Language

After the steps in "Select a Solver That Supports Code Generation" (Simulink Coder) and "Select a System Target File from STF Browser" (Simulink Coder), an optional step is to change the programming language selection for code generation. The default selection is C language for code generation.

To change the programming language setting:

1. From **Configuration Parameters** > **Code Generation** > **Language**, select `C` or `C++` for the code generation language. Alternatively, set the `TargetLang` parameter at the command line.

   The code generator produces `.c` or `.cpp` files, depending on your selection, and places the generated files in your build folder.

   For more information, see "Language" (Simulink Coder).
2. Check whether you must choose and configure a compiler. If you select C++, you must choose and configure a compiler. For details, see "Select and Configure C or C++ Compiler or IDE" (Simulink Coder).
3. Check whether the standard math library is configured for your compiler. By default, the code generator uses the ISO/IEC 9899:1999 C (`C99 (ISO)`) library for the C language and the ISO/IEC 14882:2003 C++ (`C++03 (ISO)`) library for the C++ language.

   For more information, see "Standard math library" (Simulink Coder).

More About

- "Select and Configure C or C++ Compiler or IDE" (Simulink Coder)
- "Troubleshoot Compiler Issues" (Simulink Coder)

Select and Configure C or C++ Compiler or IDE

The build process requires a supported compiler. *Compiler*, in this context, refers to a development environment (IDE) containing a linker and make utility, and a high-level language compiler. For details on supported compiler versions, see:

`http://www.mathworks.com/support/compilers/current_release`

When creating an executable program, the build process must be able to access a supported compiler. The build process can find a compiler to use based on your default MEX compiler.

The build process also requires the selection of a toolchain or template makefile. The toolchain or template makefile determines which compiler runs, during the make phase of the build. For more information, see "Choose and Configure Build Process" (Simulink Coder)

To determine which templates makefiles are available for your compiler and system target file, see "Compare System Target File Support" (Simulink Coder).

For both generated files and user-supplied files, the file extension, `.c` or `.cpp`, determines whether the build process uses a C or a C++ compiler. If the file extension is `.c`, the build process uses C compiler to compile the file, and the symbols use the C linkage convention. If the file extension is `.cpp`, the build process uses a C++ compiler to compile the file, and the symbols use the C++ linkage specification.

In this section...

Language Standards Compliance

The code generator produces code that is compliant with the following standards:

Language	Supported Standard
C	ISO/IEC 9899:1990, also known as C89/C90
C++	ISO/IEC 14882:2003

Code that the code generator produces from these sources is ANSI C/C++ compliant:

- Simulink built-in block algorithmic code
- Generated system-level code (task ID [TID] checks, management, functions, and so on)
- Code from other blocksets, including the Fixed-Point Designer product and the Communications System Toolbox product
- Code from other code generators, such as MATLAB functions

Also, the code generator can incorporate code from:

- Embedded system target files (for example, startup code, device driver blocks)
- Custom S-functions or TLC files

Note: Coding standards for these two sources are beyond the control of the code generator. These standards can be a source for compliance problems, such as code that uses C99 features not supported in the ANSI C, C89/C90 subset.

Programming Language Considerations

The code generator produces C and C++ code. Consider the following as you choose a programming language:

- Does your project require you to configure the code generator to use a specific compiler? C/C++ code generation on Windows requires this selection.
- Does your project require you to change the default language configuration setting for the model? See “Select C or C++ Programming Language” on page 40-2.
- Does your project require you to integrate legacy or custom code with generated code? For a summary of integration options, see “What Is External Code Integration?” on page 39-3.
- Does your project require you to integrate C and C++ code? If so, see “What Is External Code Integration?” on page 39-3.

Note: You can mix C and C++ code when integrating generated code with custom code. However, you must be aware of the differences between C and default C++ linkage conventions, and add the `extern "C"'` linkage specifier where required. For the details of the differing linkage conventions and how to apply `extern "C"`, refer to a C++ programming language reference book.

- Does your project require code generation support from other products? See "C++ Language Support Limitations" on page 40-5.

For C++ code generations examples with Stateflow, see the sfcndemo_cppcount model or sf_cpp model.

C++ Language Support Limitations

To use C++ language support, you could need to configure the code generator to use a specific compiler. For example, if a supported compiler is not installed on your Microsoft Windows computer, the default compiler is the `lcc` C compiler shipped with the MATLAB product. This compiler does not support C++. If you do not configure the code generator to use a C++ compiler before you specify C++ for code generation, the software produces an error message.

Code generator limitations on C++ support include:

- The code generator does not support C++ code generation for the following:
  Simscape Driveline
  Simscape Multibody First Generation (Simscape Multibody Second Generation is supported)
  Simscape Power Systems
  Simulink Real-Time
- For ERT and ERT-based system target files with **Configuration Parameters** > **Data Placement** > **Interface** > **Code interface packaging** set to `Nonreusable function`, the following fields currently do not accept the `.cpp` extension. If you specify a file name with a `.c` extension or without an extension and specify C++ for the code generation language, the code generator produces a `.cpp` file.
  - **Configuration Parameters** > **Code Placement** > **Data definition filename** field (available when **Configuration Parameters** > **Code Placement** > **Data definition** is set to `Data defined in a single separate source file`)

- **Definition file** field for a data object in Model Explorer. Data objects are objects of the class `Simulink.Signal`, `Simulink.Parameter`, and subclasses.

Code Generator Assumes Wrap on Signed Integer Overflows

The code generator reduces memory usage and enhances generated code execution by assuming signed integer C operations wrap on overflow. A signed integer overflow occurs when the result of an arithmetic operation is outside the range of values that the output data type can represent. The C programming language does not define the results of such operations. Some C compilers aggressively optimize signed operations for in-range values at the expense of overflow conditions. Other compilers preserve the full wrap-on-overflow behavior. For example, the gcc and MinGW compilers provide an option to wrap on overflow reliably for signed integer overflows. The generated program image for a model can produce results that differ from model simulation results because the handling of overflows varies, depending on your compiler.

When you generate code, if you use a supported compiler with the default options configured by the code generator, the compiler preserves the full wrap-on-overflow behavior. If you change the compiler options or compile the code in another development environment, it is possible that the compiler does not preserve the full wrap-on-overflow behavior. In this case, the executable program can produce unpredictable results.

If this issue is a concern for your application, consider one or more of the following actions:

- Verify that the compiled code produces expected results.
- If your compiler can force wrapping behavior, turn it on. For example, for the gcc compiler or a compiler based on gcc, such as MinGW, configure the build process to use the compiler option `-fwrapv`.
- Choose a compiler that wraps on integer overflow.
- If you have Embedded Coder installed, develop and apply a custom code replacement library to replace code generated for signed integers. For more information, see “Code Replacement Customization”.

Choose and Configure Compiler

The compiler for your model build appears in the build process parameters in **Configuration Parameters > Code Generation**. To view the installed compilers and select the default compiler, in the Command Window type:

```
mex -setup
```

On a Windows computer, you can install supported compilers and select a default compiler.

On a UNIX platform, the default compiler is GNU `gcc/g++` for GNU or Xcode for Mac.

Unless the build approach configuration selects a specific compiler, the code generator uses the default compiler for the build process.

Primarily, the specified system target file determines the compiler that the code generator requires:

- If you select a toolchain-based system target file such as `grt.tlc` (Generic Real-Time Target), `ert.tlc` (Embedded Coder), or `autosar.tlc` (Embedded Coder for AUTOSAR), the **Build process** subpane displays toolchain parameters for configuring the build process. Use the **Toolchain** parameter to select a compiler and associated tools for your model build. To validate the selected toolchain, click the **Validate** button for the **Configuration Parameters** > **All Parameters** > **Code Generation** > **Toolchain** box.
- If you select a template makefile (TMF) based system target file, such as `rsim.tlc`, the **Build process** subpane displays template makefile parameters for configuring the build process. The **Template makefile** parameter displays the default TMF file for the selected system target file. If the system target file supports compiler-specific template makefiles (for example, Rapid Simulation or S-Function system target files), you can set **Template makefile** to a compiler-specific TMF, such as `rsim_lcc.tmf` or `rsim_unix.tmf`. (See "Compare System Target File Support" (Simulink Coder) for valid TMF names.)

Include S-Function Source Code

When the code generator builds models with S-functions, source code for the S-functions can be either in the current folder or in the same folder as their MEX-file. The code generator adds an include path to the generated makefiles whenever it finds a file named *`sfncname`*`.h` in the same folder as the S-function MEX-file. This folder must be on the MATLAB path.

Similarly, the code generator adds a rule for the folder when it finds a file *`sfncname`*`.c` (or `.cpp`) in the same folder as the S-function MEX-file is in.

More About

- "Run-Time Environment Configuration" (Simulink Coder)
- "Compare System Target File Support" (Simulink Coder)
- "Select C or C++ Programming Language" (Simulink Coder)
- "Troubleshoot Compiler Issues" on page 40-9

External Websites

- http://www.mathworks.com/support/compilers/current_release

Troubleshoot Compiler Issues

In this section...

Compiler Version Mismatch Errors

Description

The build process produces a compiler version mismatch error.

Action

1. Check the list of supported and compatible compilers available at www.mathworks.com/support/compilers/current_release/.
2. Upgrade or change your compiler. For more information, see "Choose and Configure Compiler" on page 40-6.
3. Rebuild the model.

Results for Model Simulation and Program Execution Differ

Description

The program generated for the model produces different results from model simulation results. The generated source code includes an arithmetic operation that produces a signed integer overflow. It is possible that your compiler does not implement wrapping behavior for signed integer overflow conditions. Or, if you are using a compiler that supports wrapping, it is possible that you did not configure it to use the `-fwrapv` option.

For more information, see "Code Generator Relies on Undefined Behavior of C Language for Integer Overflows."

Action

- If your compiler can force wrapping behavior, turn it on. For example, for the gcc compiler or a compiler based on gcc, such as MinGW, specify the compiler option `-fwrapv`.
- Choose a compiler that checks for integer overflows.
- If you have Embedded Coder, develop and apply a code replacement library to replace code generated for signed integers.

Generates Expected Code and Produces Unexpected Results

Description

The build process generates expected source code, but the executable program produces unexpected results. The generated source code appears as expected. However, the executable program produces unexpected results.

Action

Do one of the following:

- Lower the compiler optimization level.
  1. Select `Custom` for the Model Configuration parameter **Code Generation** > **Compiler optimization level**.
  2. In the **Custom compiler optimization flags** field, specify a lower optimization level.
  3. Rebuild the model.
- Disable compiler optimizations.
  1. Select `Optimizations off (faster builds)` for the Model Configuration parameter **Code Generation** > **Compiler optimization level**.
  2. Rebuild the model.

For more information, see "Control Compiler Optimizations" (Simulink Coder) and your compiler documentation.

Compile-Time Issues

Issue	Action
Error is present in the compiler configuration.	Make sure that MATLAB supports the compiler and version that you want to use. For a list of currently supported and compatible compilers, see www.mathworks.com/support/compilers/current_release/. If necessary, upgrade or change your compiler (see "Choose and Configure Compiler" on page 40-6 or "Choose and Configure Compiler" on page 40-6).
Environment variables are incorrectly set up for your make utility, compiler, or linker. For example, installation of Cygwin tools on a Windows platform affects environment variables used by other compilers.	Review the environment variable settings for your system by using the `set` command on a Windows platform or `setenv` on a UNIX platform. Make sure that the settings match what is required for the tools you are using.
Error is present in custom code specified as an S-function block or in **Configuration Parameters** > **Code Generation** > **Custom Code**. For example, the code refers to a header file that the compiler cannot find.	To isolate the source of the problem, remove the custom code from the model, debug, and rebuild the model.
The model includes a block, such as a device driver block, which is not intended for use with the currently selected system target file.	Remove the system target file-specific block or configure the model for use with another system target file.
A linker error about an undefined reference to the data appears when the model build generates an executable from the model reference hierarchy and all these conditions are true: • You represent signal, state, or parameter data by creating a data object such as `Simulink.Signal`.	To resolve the issue, choose one of these methods: • In the data object, clear the `Owner` property. Alternatively, set the owner to a model that directly accesses the data. • Use a different toolchain, such as `gcc`, instead of `lcc`.

Issue	Action
You use the object in a model reference hierarchy. • You use a custom storage class with the data object. Custom storage classes require Embedded Coder. • You set the owner of the object to a model that does not directly access the data. • You use the toolchain `lcc-win64`.	

LCC Compiler Does Not Support Ampersands in Source Folder Paths

Description

If you use the LCC compiler and your model folder path contains an ampersand (`&`), the build process produces an error.

Action

Remove the ampersand from the model folder path. Then, rebuild the model.

LCC Compiler Might Not Support Line Lengths of Rapid Accelerator Code

Description

If you are compiling Rapid Accelerator code, the LCC compiler might produce an error related to line limits. Rapid Accelerator code can have longer line lengths due to obfuscation.

Action

Compile your Rapid Accelerator code using a compiler that supports longer code lines.

More About

- “Choose and Configure Compiler” on page 40-6
- “Select C or C++ Programming Language” (Simulink Coder)

- “Troubleshoot Compiler Issues” on page 40-9
- “Choose and Configure Build Process” on page 40-14
- “Build Process Workflow for a Real-Time STF” on page 40-30
- “Rebuild a Model” on page 40-46
- “Reduce Build Time for Referenced Models” on page 40-50
- “Control Regeneration of Top Model Code” on page 40-48
- “Relocate Code to Another Development Environment” on page 40-56
- “Build and Run a Program” on page 40-43
- “Profile Code Performance” on page 40-71
- “Control Compiler Optimizations” (Simulink Coder)
- “Select and Configure C or C++ Compiler or IDE” on page 40-3
- “Executable Program Generation” on page 40-68

External Websites

- www.mathworks.com/support/compilers/current_release/

Choose and Configure Build Process

The code generator supports two processes for building code generated from Simulink models:

- *Toolchain approach* — A newer build process that generates optimized makefiles and supports custom toolchains. The benefits of this approach include:
  - Provide control of your build process with toolchain information objects. You can define these objects using MATLAB scripts.
  - Supports model referencing, PIL, SIL, and Rapid Accelerator builds.
  - Supported by MATLAB Coder.
  - Provides added flexibility to configure the build process for individual models.
- *Template makefile approach* — An older build process that uses template makefiles

The "System target file" (Simulink Coder) parameter, located in the **Configuration Parameters > Code Generation** pane, determines the build process for a model. When the **System target file** is set to:

- `ert.tlc`, `ert_shrlib.tlc`, `grt.tlc`, or any toolchain-compliant system target file, the build process uses the *toolchain approach*. For more information, see "Toolchain Approach" on page 40-14 and see "Support Toolchain Approach with Custom Target" on page 71-81.
- Any non-toolchain-compliant system target files. The build process uses the *template makefile approach*. For more information, see "Template Makefile Approach" on page 40-20.

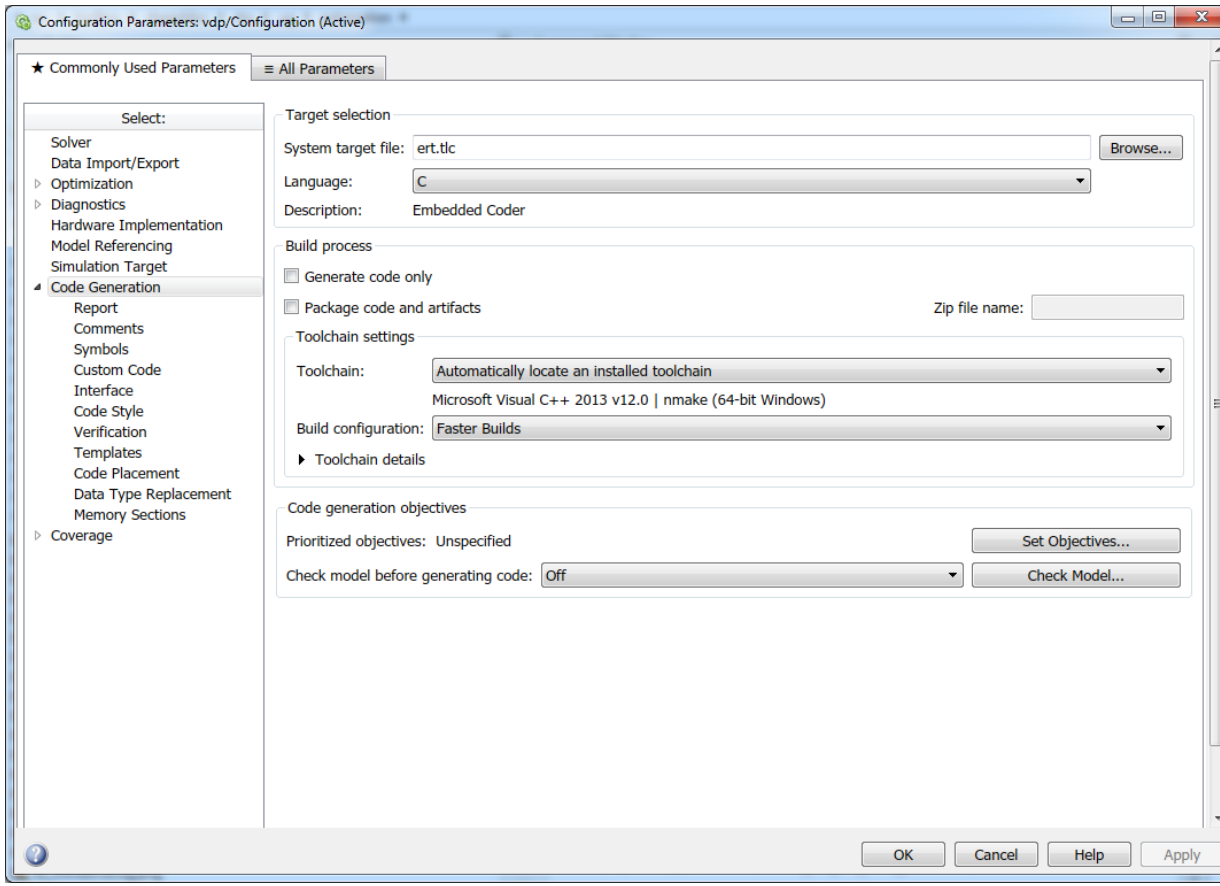
In this section...

Toolchain Approach

The *toolchain approach* is named for the **Toolchain settings** that appear under **Build process** when you set **System target file** to:

- `grt.tlc` — `Generic Real-Time Target`
- `ert.tlc` — `Embedded Coder` (requires the Embedded Coder product)
- `ert_shrlib.tlc` — `Embedded Coder (host-based shared library target)` (requires the Embedded Coder product)
- Any toolchain-compliant system target file (If ERT-based, requires the Embedded Coder product)

For more information about toolchain-compliant system target files, see "Support Toolchain Approach with Custom Target" (Simulink Coder).



The **Toolchain settings** include:

- The "Toolchain" (Simulink Coder) parameter specifies the collection of third-party software tools that builds the generated code. A toolchain can include a compiler, linker, archiver, and other prebuild or postbuild tools that download and run the executable on the target hardware.

  The default value of **Toolchain** is `Automatically locate an installed toolchain`. The **Toolchain** parameter displays name of the located toolchain just below `Automatically locate an installed toolchain`.

  Click the **Validate** button for the **Configuration Parameters** > **All Parameters** > **Code Generation** > **Toolchain** parameter to check that the toolchain is present and validate that the code generator has the information required to use the toolchain. The resulting Validation Report gives a pass/fail for the selected toolchain, and identifies issues to resolve.
- The "Build configuration" (Simulink Coder) parameter lets you choose or customize the optimization settings. By default, **Build Configuration** is set to `Faster Builds`. You can also select `Faster Runs`, `Debug`, and `Specify`. When you select `Specify` and click **Apply**, you can customize the toolchain options for each toolchain. These custom toolchain settings only apply to the current model.

Note: The following system target files, which use the template makefile approach, have the same names but different descriptions from system target files that use the toolchain approach:

- `ert.tlc – Create Visual C/C++ Solution File for Embedded Coder`
- `grt.tlc – Create Visual C/C++ Solution File for Simulink Coder`

To avoid confusion, click **Browse** to select the system target file and look at the description of each file.

Upgrade Model to Use Toolchain Approach

When you open a model created before R2013b that uses the following system target files, the software tries to upgrade the model. The upgrade changes the configuration from using template makefile settings to using the toolchain settings:

- `ert.tlc – Embedded Coder`
- `ert_shrlib.tlc – Embedded Coder (host-based shared library target)`

- `grt.tlc — Generic Real-Time Target`

Note: To upgrade models using a custom system target file to use the toolchain approach, see "Support Toolchain Approach with Custom Target" on page 71-81.

Some model configuration parameter values prevent the software from upgrading a model to use toolchain settings. The following instructions show you ways to complete the upgrade process.

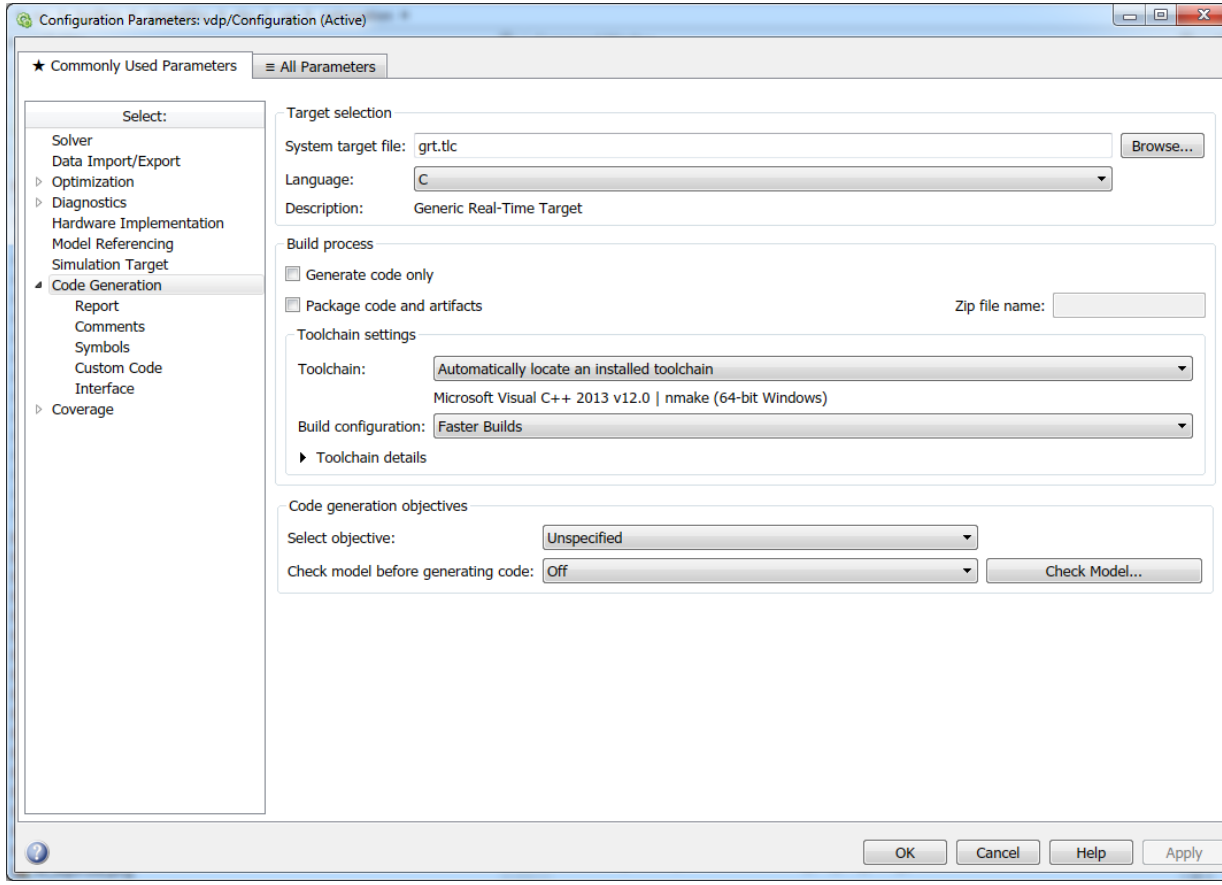
Consider upgrading your models and use the toolchain build approach. Doing so is not required. You can continue generating code from a model that has not been upgraded.

Note: The software does not upgrade models that use the following system target files:

- `ert.tlc — Create Visual C/C++ Solution File for Embedded Coder`
- `grt.tlc — Create Visual C/C++ Solution File for Simulink Coder`

To see if a model was upgraded:

1. Open the model configuration parameters by pressing **Ctrl+E**.
2. Select **Configuration Parameters** > **Code Generation**.
3. If the **Build process** subpane contains the **Toolchain** and **Build configuration** parameters, the model has already been upgraded.



If the **Build process** area displays **Makefile configuration** parameters, such as **Generate makefile**, **Make command**, and **Template makefile**, the software has not upgraded the model.

Start by creating a working copy of the model using **File** > **Save As**. This action preserves the original model and configuration parameters for reference.

Try to upgrade the model using Upgrade Advisor:

1 In your model, select **Analysis** > **Model Advisor** > **Upgrade Advisor**.

2. In Upgrade Advisor, select **Check and update model to use toolchain approach to build generated code** and click **Run This Check**.
3. Perform the suggested actions and/or click **Update Model**.

When you cannot upgrade the model using Upgrade Advisor, one or more of the following parameters is not set to its default value, shown here:

- **Compiler optimization level** — `Optimizations off (faster builds)`
- **Generate makefile** — Enabled
- **Template makefile** — System target file-specific template makefile
- **Make command** — `make_rtw` without arguments

Sometimes, a model cannot be upgraded. Try the following procedure:

- If **Generate makefile** is disabled, this case cannot be upgradable. However, you can try enabling it and try upgrading the model using Upgrade Advisor.
- If **Compiler optimization level** is set to `Optimizations on (faster runs)`:

  1. Set **Compiler optimization level** is to `Optimizations off (faster builds)`.
  2. Upgrade the model using Upgrade Advisor.
  3. Set **Build configuration** to `Faster Runs`.
- If **Compiler optimization level** is set to `Custom`:

  1. Copy the **Custom compiler optimization flags** to a text file.
  2. Set **Compiler optimization level** to `Optimizations off (faster builds)`.
  3. Upgrade the model using Upgrade Advisor.
  4. Set **Build configuration** to `Specify`.
  5. To perform the same optimizations, edit the compiler options.
- If **Template makefile** uses a customized template makefile, this case cannot be upgradable. However, you can try the following:

  1. Update **Template makefile** to use the default makefile for the system target file.

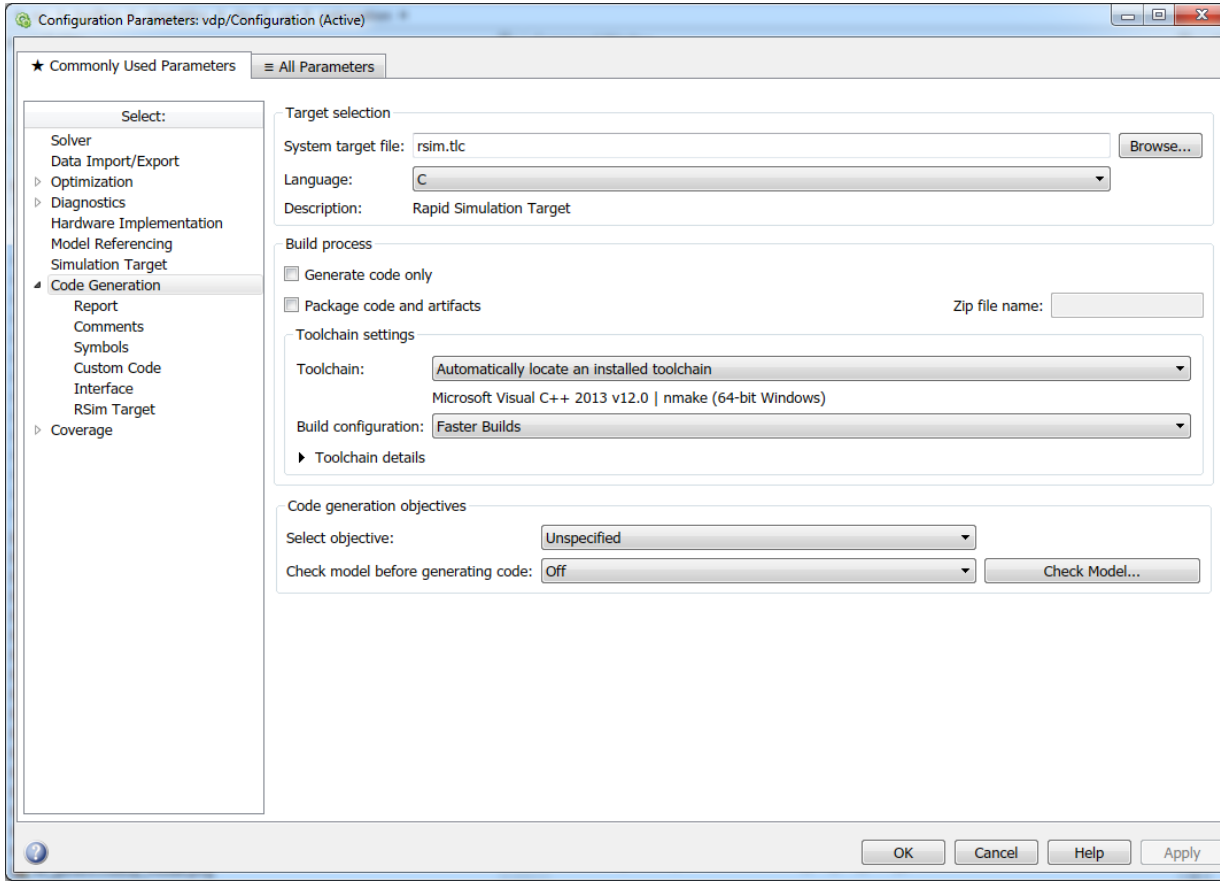
     Note: To get the default makefile name, change the **System target file**, click **Apply**, change it back, and click **Apply** again.

2. Upgrade the model using Upgrade Advisor.
3. If the template makefile contains build tool options, such as compiler optimization flags, set **Build configuration** to `Specify` and update the options.
4. If the template makefile uses custom build tools, create and register a custom toolchain, as described in "Custom Toolchain Registration" (MATLAB Coder) . Then, set the **Toolchain** parameter to use the custom toolchain.

   Note: After registering the custom toolchain, update **Toolchain** to use the custom toolchain.
5. If the template makefile contains custom rules and logic, these customizations cannot be applied to the upgraded model.

Template Makefile Approach

When the **System target file** is set to a `tlc` file that uses the template makefile approach, the software displays **Compiler optimization level**, **Generate makefile**, **Make command**, and **Template makefile** parameters.



Specify Whether to Generate a Makefile

The **Generate makefile** option specifies whether the build process is to generate a makefile for a model. By default, the build process generates a makefile. Suppress generation of a makefile, for example in support of custom build processing that is not based on makefiles, by clearing **Generate makefile**. When you clear this parameter:

- The **Make command** and **Template makefile** options are unavailable.
- Set up post code generation build processing using a user-defined command, as explained in "Customize Post-Code-Generation Build Processing" (Simulink Coder).

Specify a Make Command

Each template makefile-based system target file has an associated `make` command. The code generator uses this internal MATLAB command to control the build process. The command appears in the **Make command** field and runs when you start a build.

Most system target files use the default command, `make_rtw`. Third-party system target files could supply another `make` command. See the documentation from the vendor.

In addition to the name of the `make` command, you can supply makefile options in the **Make command** field. These options could include compiler-specific options, include paths, and other parameters. When the build process invokes the `make` utility, these options are passed on the `make` command line, which adds them to the overall flags passed to the compiler.

"Template Makefiles and Make Options" on page 40-24 lists the **Make command** options you can use with each supported compiler.

Specify the Template Makefile

The **Template makefile** field has these functions:

- If you selected a system target file with the System Target File Browser, this field displays the name of a MATLAB language file that selects a template makefile for your development environment. For example, in "Model Configuration Parameters: Code Generation" (Simulink Coder), the **Template makefile** field displays `grt_default_tmf`, indicating that the build process invokes `grt_default_tmf.m`.

  "Template Makefiles and Make Options" on page 40-24 gives a detailed description of the logic by which the build process selects a template makefile.
- Alternatively, you can explicitly enter the name of a specific template makefile (including the extension) or a MATLAB language file that returns a template makefile in this field. Use this approach if you are using a system target file that does not appear in the System Target File Browser. For example, use this approach if you have written your own template makefile for a custom system target file.

If you specify your own template makefile, be sure to include the file name extension. If you omit the extension, the build process attempts to find and execute a file with the extension `.m` (that is, a MATLAB language file). The template make file (or a MATLAB language file that returns a template make file) must be on the MATLAB path. To determine whether the file is on the MATLAB path, enter the following command in the MATLAB Command Window:

```
which tmf_filename
```

Specify TLC for Code Generation

You can specify Target Language Compiler (TLC) command-line options and arguments for code generation using the model parameter `TLCOptions` in a `set_param` function call. For example,

```
>> set_param(gcs,'TLCOptions','-p0 -aWarnNonSaturatedBlocks=0')
```

Some common uses of TLC options include the following:

- `-aVarName=1` to declare a TLC variable and/or assign a value to it
- `-IC:\Work` to specify an include path
- `-v` to obtain verbose output from TLC processing (for example, when debugging)

TLC options that you specify for code generation appear in the summary section of the generated HTML code generation report.

Specifying TLC command-line options does not add flags to the make command line.

For additional information, see "Target Language Compiler Overview" (Simulink Coder).

More About

- "Support Toolchain Approach with Custom Target" on page 71-81
- "Build and Run a Program" on page 40-43
- "Custom Toolchain Registration" (MATLAB Coder)
- "Template Makefiles and Make Options" on page 40-24
- "Target Language Compiler Overview" (Simulink Coder)
- "Executable Program Generation" on page 40-68

Template Makefiles and Make Options

The code generator includes a set of built-in template makefiles that build programs for specific system target files.

In this section...

Types of Template Makefiles

There are two types of template makefiles:

- *Compiler-specific* template makefiles are for a particular compiler or development system.

  By convention, compiler-specific template makefiles names correspond to the system target file and compiler (or development system). For example, `grt_vcx64.tmf` is the template makefile for building a generic real-time program under the Visual C++ compiler; `ert_lcc.tmf` is the template makefile for building an Embedded Coder program under the `lcc` compiler.
- *Default* template makefiles make your model designs more portable, by choosing the compiler-specific makefile and compiler for your installation. "Select and Configure C or C++ Compiler or IDE" on page 40-3 describes the operation of default template makefiles in detail.

  Default template makefiles have names that follow the pattern *`target`*`_default_tmf`. They are MATLAB language files that, when run, select the TMF for the specified system target file configuration. For example, `grt_default_tmf` is the default template makefile for building a generic real-time program; `ert_default_tmf` is the default template makefile for building an Embedded Coder program.

For details on the structure of template makefiles, see "Customize Template Makefiles" on page 71-62. This section describes compiler-specific template makefiles and common options you can use with each.

Specify Template Makefile Options

You can specify template makefile options by using the **Make command** box in **Configuration Parameters** > **Code Generation**. Append the options after `make_rtw` (or other `make` command), as in the following example:

```
make_rtw OPTS="-DMYDEFINE=1"
```

The syntax for `make` command options differs slightly for different compilers.

Note: To control compiler optimizations for a makefile build at the Simulink GUI level, use the **Compiler optimization level** parameter on the **All Parameters** tab of the Configuration Parameters dialog box. The **Compiler optimization level** parameter provides

- System target file-independent values `Optimizations on (faster runs)` and `Optimizations off (faster builds)`, which easily allow you to toggle compiler optimizations on and off during code development
- The value `Custom` for entering custom compiler optimization flags at Simulink GUI level (rather than at other levels of the build process)

If you specify compiler options for your makefile build using `OPT_OPTS`, `MEX_OPTS` (except `MEX_OPTS="-v"`), or `MEX_OPT_FILE`, the value of **Compiler optimization level** is ignored and a warning is issued about the ignored parameter.

Template Makefiles for UNIX Platforms

The template makefiles for UNIX platforms are for the Free Software Foundation's GNU Make. These makefiles conform to the guidelines specified in the IEEE®[10] Std 1003.2-1992 (POSIX) standard.

- `ert_unix.tmf`
- `grt_unix.tmf`
- `rsim_unix.tmf`
- `rtwsfcn_unix.tmf`

10. IEEE is a registered trademark of The Institute of Electrical and Electronics Engineers, Inc.

You can supply options to makefiles using the **Make command** box in the **Configuration Parameters > Code Generation** pane. Options specified in **Make command** are passed to the command-line invocation of the `make` utility, which adds them to the overall flags passed to the compiler. The following options can be used to modify the behavior of the build:

- `OPTS` — User-specific options, for example,

  ```
  OPTS="-DMYDEFINE=1"
  ```

- `OPT_OPTS`— Optimization options. Default is `-O`. To enable debugging, specify the option as `OPT_OPTS=-g`. Because of optimization problems in IBM_RS, the default is no optimization.
- `CPP_OPTS` — C++ compiler options.
- `USER_SRCS` — Additional user sources, such as files used by S-functions.
- `USER_INCLUDES` — Additional include paths, for example,

  ```
  USER_INCLUDES="-Iwhere-ever -Iwhere-ever2"
  ```

- `DEBUG_BUILD` — Add debug information to generated code, for example,

  ```
  DEBUG_BUILD=1
  ```

These options are also documented in the comments at the head of the respective template makefiles.

Template Makefiles for the Microsoft Visual C++ Compiler

Visual C++ Executable Build

To build an executable using the Visual C++ compiler within the build process, use one of the *target*`_vcx64.tmf` template makefiles:

- `ert_vcx64.tmf`
- `grt_vcx64.tmf`
- `rsim_vcx64.tmf`
- `rtwsfcn_vcx64.tmf`

You can supply options to makefiles using the **Make command** field in the **Configuration Parameters > Code Generation** pane. Options specified in **Make command** are passed to the command-line invocation of the `make` utility, which adds them to the overall flags passed to the compiler. The following options can be used to modify the behavior of the build:

- `OPT_OPTS` — Optimization option. Default is `-O2`. To enable debugging, specify the option as `OPT_OPTS=-Zi`.
- `OPTS` — User-specific options.
- `CPP_OPTS` — C++ compiler options.
- `USER_SRCS` — Additional user sources, such as files used by S-functions.
- `USER_INCLUDES` — Additional include paths, for example,

  ```
  USER_INCLUDES="-Iwhere-ever -Iwhere-ever2"
  ```

- `DEBUG_BUILD` — Add debug information to generated code, for example,

  ```
  DEBUG_BUILD=1
  ```

These options are also documented in the comments at the head of the respective template makefiles.

Visual C++ Code Generation Only

To create a Visual C++ project makefile (*`model`*`.mak`) without building an executable, use one of the *`target`*`_msvc.tmf` template makefiles:

- `ert_msvc.tmf`
- `grt_msvc.tmf`

These template makefiles are for `nmake`, which is bundled with the Visual C++ compiler.

You can supply options to makefiles using the **Make command** field in the **Configuration Parameters > Code Generation** pane. Options specified in **Make command** are passed to the command-line invocation of the `make` utility, which adds them to the overall flags passed to the compiler. The following options can be used to modify the behavior of the build:

- `OPTS` — User-specific options, for example,

  ```
  OPTS="/D MYDEFINE=1"
  ```

- `USER_SRCS` — Additional user sources, such as files used by S-functions.
- `USER_INCLUDES` — Additional include paths, for example,

  ```
  USER_INCLUDES="-Iwhere-ever -Iwhere-ever2"
  ```

- `DEBUG_BUILD` — Add debug information to generated code, for example,

  ```
  DEBUG_BUILD=1
  ```

These options are also documented in the comments at the head of the respective template makefiles.

Template Makefiles for the LCC Compiler

The code generator provides template makefiles to create an executable for the Windows platform using Lcc compiler Version 2.4 and GNU Make (`gmake`).

- `ert_lcc.tmf`
- `grt_lcc.tmf`
- `rsim_lcc.tmf`
- `rtwsfcn_lcc.tmf`

You can supply options to makefiles using the **Make command** field in the **Configuration Parameters > Code Generation** pane. Options specified in **Make command** are passed to the command-line invocation of the `make` utility, which adds them to the overall flags passed to the compiler. The following options can be used to modify the behavior of the build:

- `OPTS` — User-specific options, for example,

  ```
  OPTS="-DMYDEFINE=1"
  ```

- `OPT_OPTS` — Optimization options. Default is no options. To enable debugging, specify `-g4`:

  ```
  OPT_OPTS="-g4"
  ```

- `CPP_OPTS` — C++ compiler options.
- `USER_SRCS` — Additional user sources, such as files used by S-functions.
- `USER_INCLUDES` — Additional include paths. For example:

  ```
  USER_INCLUDES="-Iwhere-ever -Iwhere-ever2"
  ```

For `lcc`, use `/` as file separator before the file name instead of `\`, for example, `d:\work\proj1/myfile.c`.

- `DEBUG_BUILD` — Add debug information to generated code, for example,

  ```
  DEBUG_BUILD=1
  ```

These options are also documented in the comments at the head of the respective template makefiles.

More About

- "Select a System Target File" on page 30-2
- "Customize System Target Files" (Simulink Coder)

Build Process Workflow for a Real-Time STF

Building a program means generating C or C++ code from an example model and then building an executable program from the generated code. This example can use a generic real-time (GRT) or an embedded real-time (ERT) system target file (STF) for code generation. The resulting standalone program runs on your desktop computer, independent of external timing and events.

In this section...

Working Folder

This example uses a local copy of the `slexAircraftExample` model, stored in its own folder, `aircraftexample`. Set up your *working folder* as follows:

1. In the MATLAB Current Folder browser, navigate to a folder to which you have write access.
2. To create the working folder, enter the following MATLAB command:

   ```
   mkdir aircraftexample
   ```
3. Make `aircraftexample` your working folder:

   ```
   cd aircraftexample
   ```
4. Open the `slexAircraftExample` model:

   ```
   slexAircraftExample
   ```

   The model appears in the Simulink Editor model window.

5 In the model window, choose **File** > **Save As**. Navigate to your working folder, `aircraftexample`. Save a copy of the `slexAircraftExample` model as `myAircraftExample`.

Build Folder and Code Generation Folders

While producing code, the code generator creates a *build folder* within your working folder. The build folder name is *`model_target`*`_rtw`, derived from the name of the source model and the chosen system target file. The build folder stores generated source code and other files created during the build process. Examine the build folder contents at the end of this example.

When a model contains Model blocks (references to other models), the model build creates special subfolders in your "Code generation folder" (Simulink) to organize code for the referenced models. These code generation folders exist alongside product build folders and are named `slprj`. "Generate Code for Referenced Models" on page 5-4 describes navigating code generation folder structures in Model Explorer.

Under the `slprj` folder, a subfolder named `_sharedutils` contains generated code that can be shared between models.

Set Simulation Parameters

To generate code from your model, you must change some of the simulation parameters. In particular, the generic real-time (GRT) system target file and most other system target files require that the model specifies a fixed-step solver.

Note The code generator produces code for models, using variable-step solvers, for rapid simulation (`rsim`) and S-function system target files only.

To set simulation parameters using the Configuration Parameters dialog box:

1 Open the `myAircraftExample` model if it is not already open.

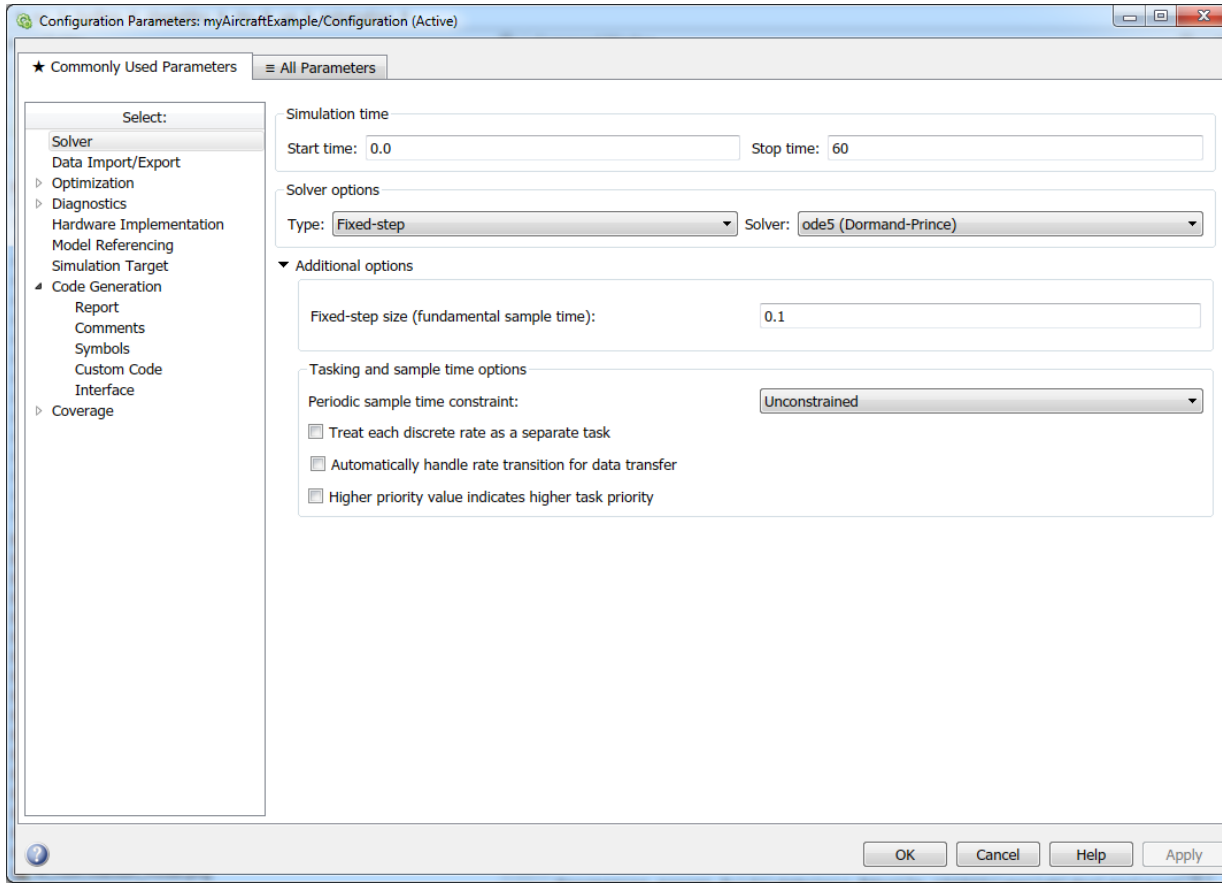
2 From the model window, open the Configuration Parameters dialog box by selecting **Simulation** > **Model Configuration Parameters**.

3 Select **Configuration Parameters** > **Solver**. Enter the following parameter values (some could already be set):

- **Start time**: `0.0`

- **Stop time**: `60`
- **Type**: `Fixed-step`
- **Solver**: `ode5 (Dormand-Prince)`
- **Fixed step size (fundamental sample time)**: `0.1`
- **Treat each discrete rate as a separate task**: `Off`

The background color of the controls you just changed is tan. The color also appears on fields as a reaction to your choices in other fields. Use this visual feedback to verify that what you set is what you intended. When you apply your changes, the background color reverts to white.



4 To register your changes, click **Apply**.

5 Save the model. Simulation parameters persist with the model for use in future sessions.

Configure Build Process

To configure the build process for your model, choose a system target file, a toolchain or template makefile, and a `make` command.

In these examples and in most applications, you do not need to specify these parameters individually. The examples use the ready-to-run generic real-time target (GRT) configuration. The GRT system target file builds a standalone executable program that runs on your desktop computer.

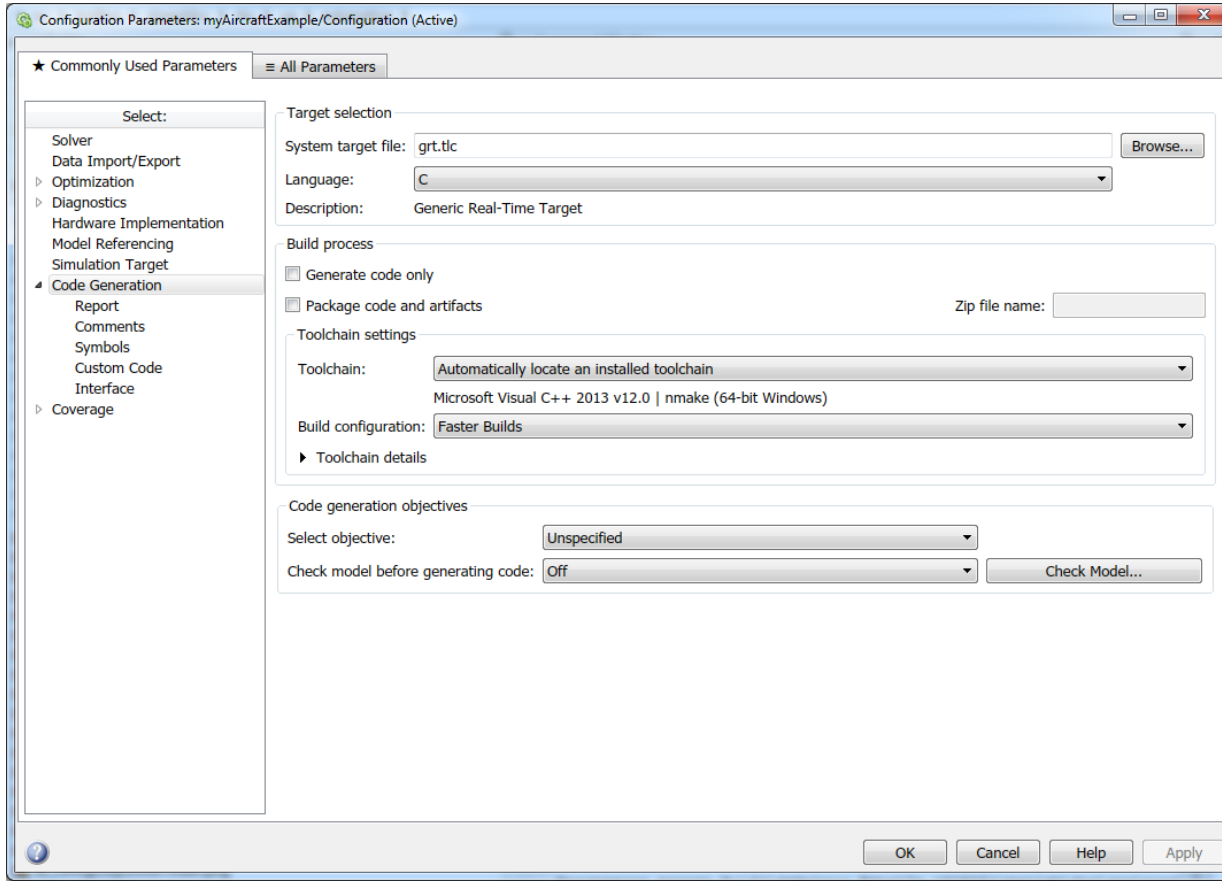
To select the GRT system target file using the Configuration Parameters dialog box:

1 With the `myAircraftExample` model open, select **Simulation** > **Model Configuration Parameters** to open the Configuration Parameters dialog box.

2 Select **Configuration Parameters** > **Code Generation**.

3 For **System target file**, enter `grt.tlc`, and click **Apply**.

The **Configuration Parameters** > **Code Generation** pane displays selections for the **System target file** (`grt.tlc`), **Toolchain** (`Automatically locate an installed toolchain`), and **Build Configuration** (`Faster Builds`).



Note If you click **Browse**, a System Target File Browser opens and displays the system target files on the MATLAB path. Some system target files require additional products. For example, `ert.tlc` requires Embedded Coder.

4 Save the model.

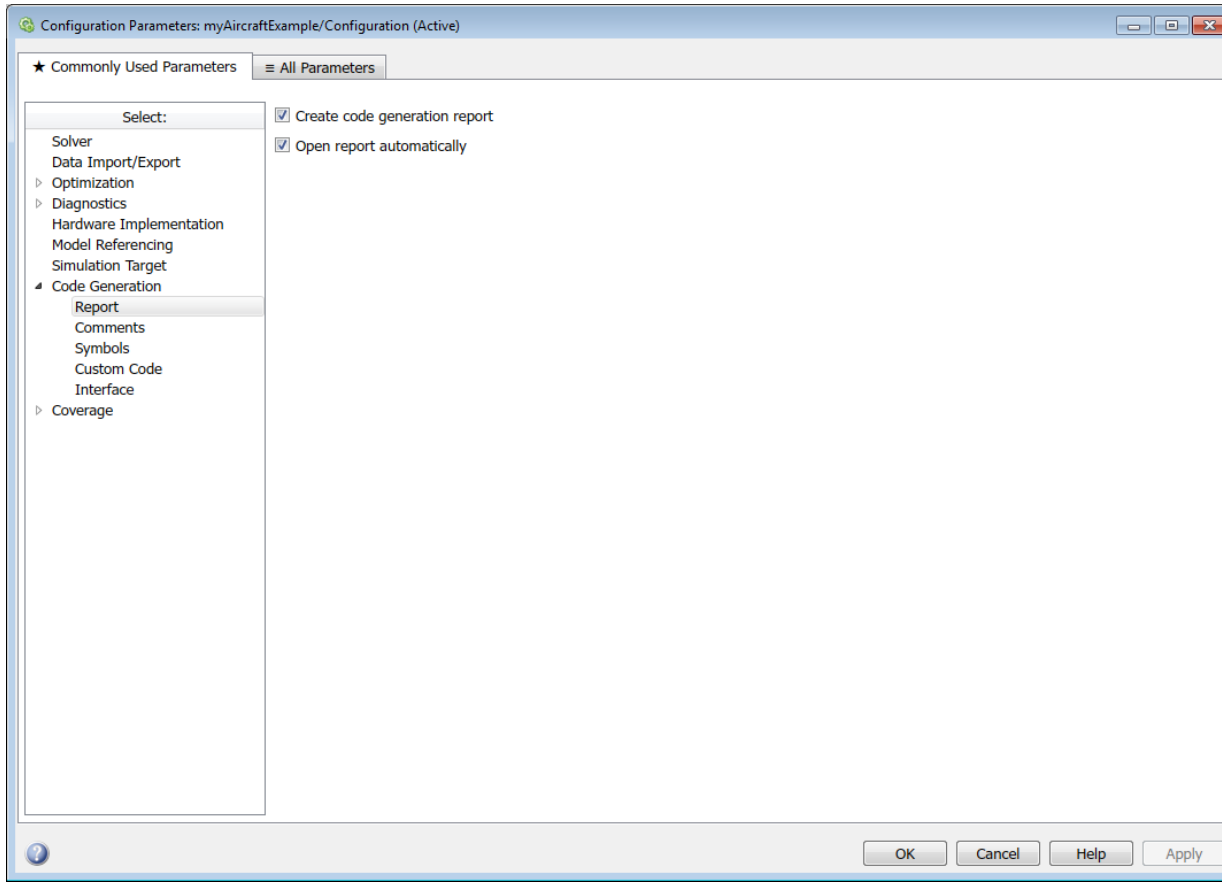
Set Code Generation Parameters

Before you generate code from your model for the first time, inspect the code generation parameters for the model. Some of the steps in this topic do not require you to change

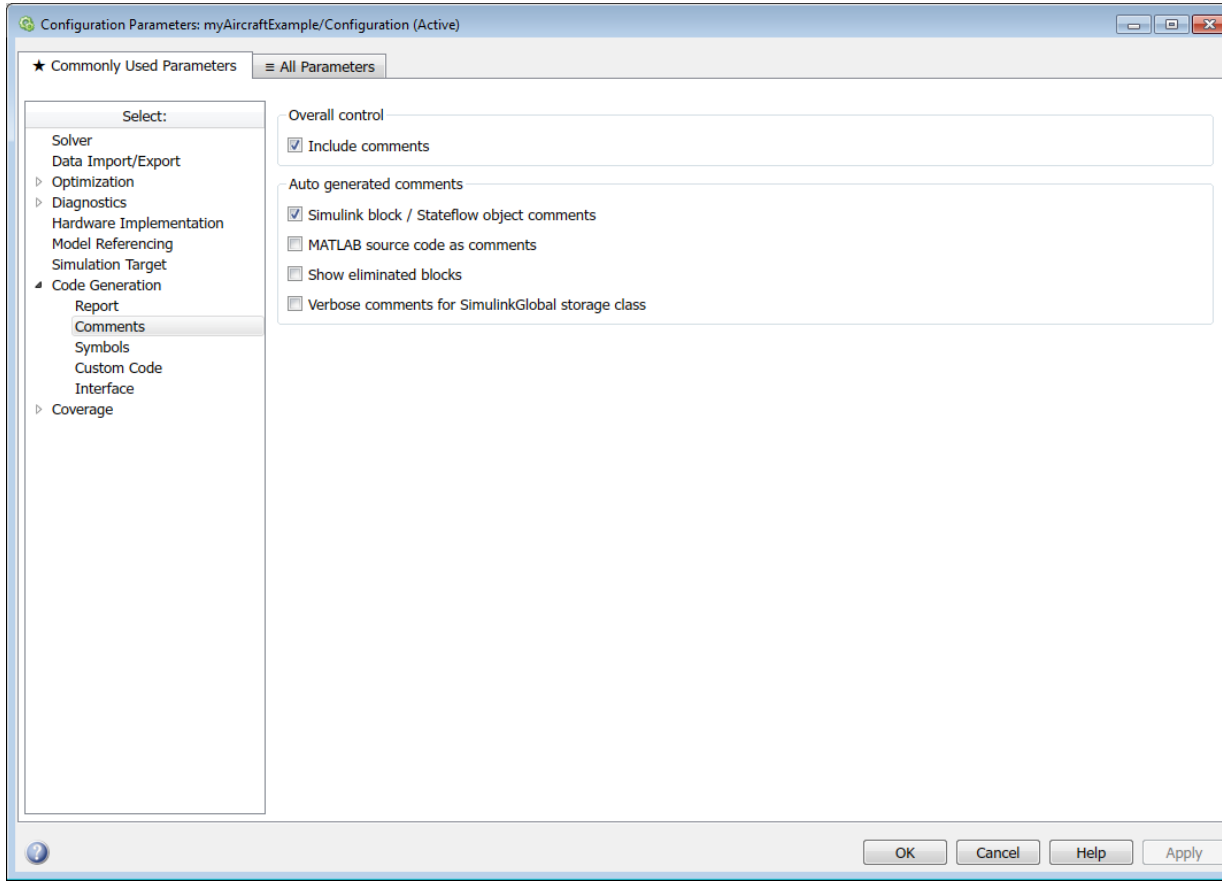
parameter values. These steps appear to help you familiarize yourself with the user interface. As you work with the model parameters, you can place the mouse pointer on a parameter to see a tool tip describing its function.

To set code generation parameters using the Configuration Parameters dialog box:

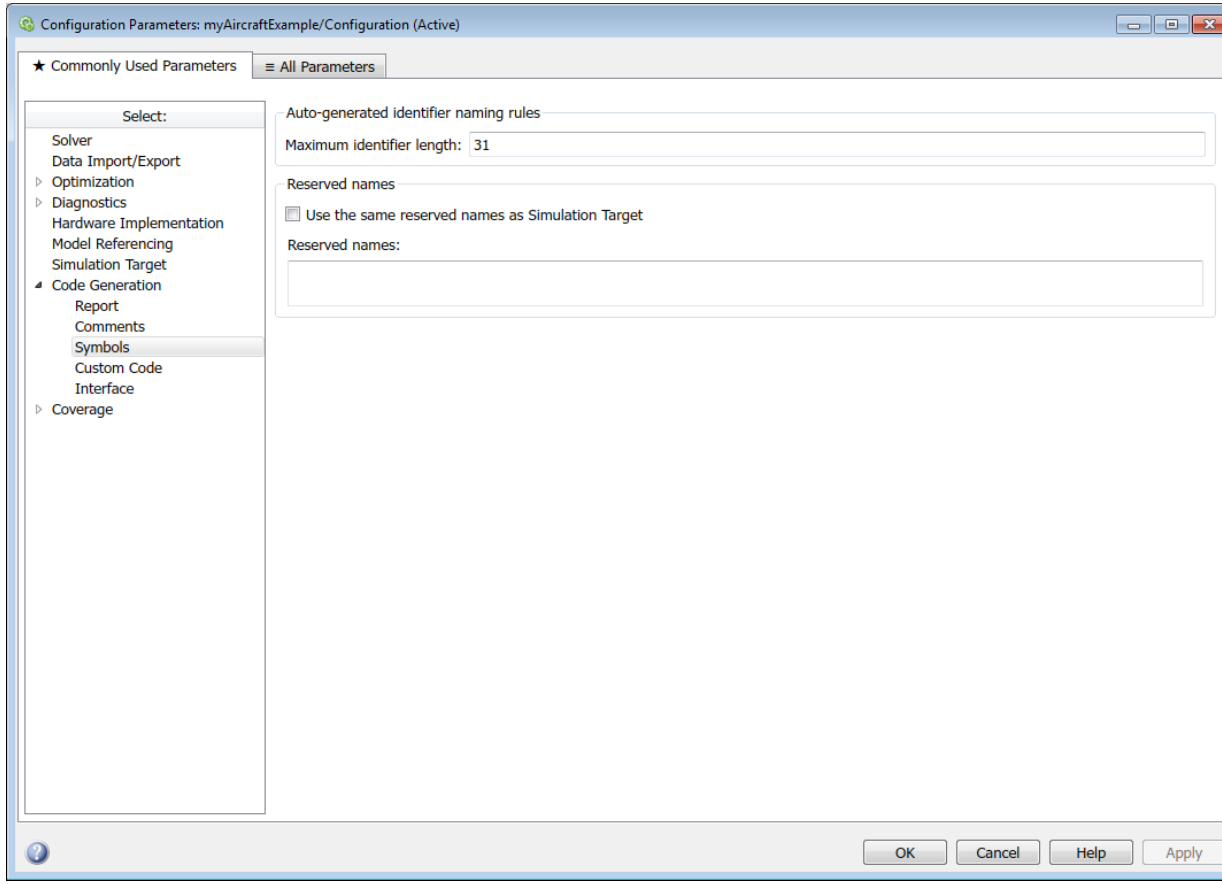
1. With the `myAircraftExample` model open, select **Simulation** > **Model Configuration Parameters** to open the Configuration Parameters dialog box.
2. Select **Code Generation** > **Report** > **Create code generation report**. This action enables the software to create and display a code generation report for the `myAircraftExample` model.



3 Select **Code Generation** > **Comments**. The options displayed here control the types of comments included in generated code. Leave the options set to their defaults.



4 Select **Code Generation** > **Symbols**. These options control the look and feel of generated code.



5 Select the **All Parameters** tab. The following parameters in the **Code Generation** category control build verbosity and debugging support. These parameters are common to all system target file configurations.

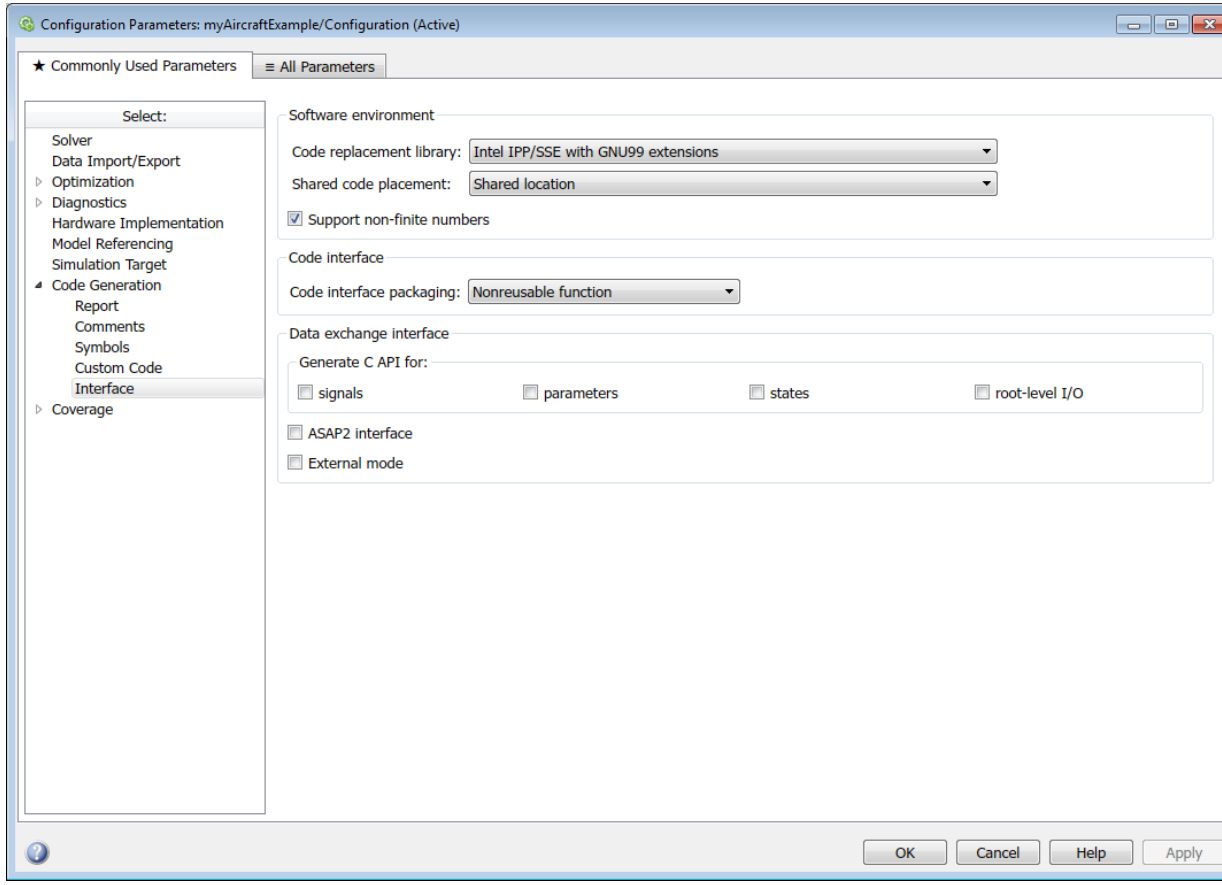
- **Verbose build** (`RTWVerbose` parameter)
- **Retain .rtw file** (`RetainRTWFile` parameter)
- **Profile TLC** (`ProfileTLC` parameter)
- **Start TLC debugger when generating code** (`TLCDebug` parameter)
- **Start TLC coverage when generating code** (`TLCCoverage` parameter)

- **Enable TLC assertion** (`TLCAssert` parameter)

Leave the options set to their defaults.

6 Select **Configuration Parameters** > **Code Generation** > **Interface**.

- For the **Shared code placement** parameter, select `Shared location`. The build process places generated code for utilities at a shared location within the `slprj` folder in your "Code generation folder" (Simulink).
- If it is not already cleared, switch to the **All Parameters** tab and clear the **Code Generation** > **Classic call interface** option.



7 Click **Apply** and save the model.

Build and Run a Program

The build process generates C code from the model. It then compiles and links the generated program to create an executable image. To build and run the program:

1 With the `myAircraftExample` model open, initiate code generation and the build process for the model by using any of the following options:

- Click the **Build Model** button.
- Press **Ctrl+B**.
- Select **Code** > **C/C++ Code** > **Build Model**.
- Invoke the `rtwbuild` command from the MATLAB command line.
- Invoke the `slbuild` command from the MATLAB command line.

Some messages concerning code generation and compilation appear in the Command Window. The initial message is:

```
### Starting build procedure for model: myAircraftExample
```

The contents of many of the succeeding messages depends on your compiler and operating system. The final messages include:

```
### Created executable myAircraftExample.exe
### Successful completion of build procedure for model: myAircraftExample
### Creating HTML report file myAircraftExample_codegen_rpt.html
```

The code generation folder now contains an executable, `myAircraftExample.exe` (Microsoft Windows platforms) or `myAircraftExample` (UNIX platforms). In addition, the build process has created an `slprj` folder and a `myAircraftExample_grt_rtw` folder in your "Code generation folder" (Simulink).

Note: After generating the code for the `myAircraftExample` model, the build process displays a code generation report. See "Report Generation" (Simulink Coder) for more information about how to create and use a code generation report.

---

2 To see the contents of the working folder after the build, enter the `dir` or `ls` command:

```
>> dir
```

```
.                              myAircraftExample.slx           slprj
..                             myAircraftExample.slx.autosave
myAircraftExample.exe          myAircraftExample_grt_rtw
```

3 To run the executable from the Command Window, type `!myAircraftExample`. The `!` character passes the command that follows it to the operating system, which runs the standalone `myAircraftExample` program.

```
>> !myAircraftExample

** starting the model **
** created myAircraftExample.mat **
```

4 To see the files created in the build folder, use the `dir` or `ls` command again. The exact list of files produced varies among MATLAB platforms and versions. Here is a sample list from a Windows platform:

```
>> dir myAircraftExample_grt_rtw

.                     rt_main.obj                 myAircraftExample_data.c
..                    rtmodel.h                   myAircraftExample_data.obj
buildInfo.mat         rtw_proj.tmw                myAircraftExample_private.h
codeInfo.mat          myAircraftExample.bat       myAircraftExample_ref.rsp
defines.txt           myAircraftExample.c         myAircraftExample_types.h
html                  myAircraftExample.h
modelsources.txt      myAircraftExample.mk
rt_logging.obj        myAircraftExample.obj
```

Contents of the Build Folder

The build process creates a build folder and names it *`model`*`_`*`target`*`_rtw`, where *`model`* is the name of the source model and *`target`* is the system target file selected for the model. In this example, the build folder is named `myAircraftExample_grt_rtw`.

The build folder includes the following generated files.

File	Description
`myAircraftExample.c`	Standalone C code that implements the model
`myAircraftExample.h`	An include header file containing definitions of parameters and state variables
`myAircraftExample_private.h`	Header file containing common include definitions
`myAircraftExample_types.h`	Forward declarations of data types used in the code

File	Description
`rtmodel.h`	Master header file for including generated code in the static main program (its name does not change, and it simply includes `myAircraftExample.h`)

The code generation report that you created for the `myAircraftExample` model displays a link for each of these files. You can click the link explore the file contents.

The build folder contains other files used in the build process. They include:

- `myAircraftExample.mk` — Makefile for building executable using the specified Toolchain.
- Object (`.obj`) files
- `myAircraftExample.bat` — Batch control file
- `rtw_proj.tmw` — Marker file
- `buildInfo.mat` — Build information for relocating generated code to another development environment
- `defines.txt` — Preprocessor definitions required for compiling the generated code
- `myAircraftExample_ref.rsp` — Data to include as command-line arguments to `mex` (Windows systems only)

The build folder also contains a subfolder, `html`, which contains the files that make up the code generation report. For more information, see "Reports for Code Generation" (Simulink Coder).

Customized Makefile Generation

After producing code, the code generator produces a customized makefile, *`model`*`.mk`. The generated makefile instructs the `make` system utility to compile and link source code generated from the model, any required harness program, libraries, or user-provided modules. The code generator produces the file *`model`*`.mk` whether you are using the toolchain approach or template makefile approach for build process control. For more information about these approaches, see "Choose and Configure Build Process" on page 40-14.

If you are using the toolchain approach, the code generator creates *`model`*`.mk` based on the model **Toolchain settings**. You can modify generation of the makefile through the `rtwmakecfg.m` API. For more information, see "Toolchain Approach" on page 40-14.

If you are using the template makefile approach, the code generator creates *model*.mk from a system template file, *system*.tmf (where *system* stands for the selected system target file name). The system template makefile is designed for your system target file. You have the option of modifying the template makefile to specify compilers, compiler options, and additional information used during the creation of the executable. For more information, see "Template Makefile Approach" on page 40-20.

More About

- "Choose and Configure Build Process" on page 40-14
- "Build and Run a Program" on page 40-43
- "Reports for Code Generation" (Simulink Coder)
- "Select and Configure C or C++ Compiler or IDE" on page 40-3

Build and Run a Program

The build process generates C code from a model, then compiles and links the generated program to create an executable image. To build and run a sample program, use the example model `slexAircraftExample`.

1. In the Command Window, enter `slexAircraftExample` to open the model.
2. Save a copy of the model to your working folder and name it `myAircraftExample`.
3. Open the Configuration Parameters dialog box by selecting **Simulation** > **Model Configuration Parameters**. Set the following parameters.

   - Select **Configuration Parameters** > **Solver**. Enter the following parameter values for the **Solver** (some could already be set):

     - **Start time**: `0.0`
     - **Stop time**: `60`
     - **Type**: `Fixed-step`
     - **Solver**: `ode5 (Dormand-Prince)`
     - **Fixed step size (fundamental sample time)**: `0.1`
     - **Treat each discrete rate as a separate task**: `Off`

   - Select **Configuration Parameters** > **Code Generation** > **Report**, and select the **Create code generation report** parameter. This option causes the build process to display a code generation report after generating the code for the `myAircraftExample` model.
   - Select **Configuration Parameters** > **Code Generation** > **Interface**. For the **Shared code placement** parameter, select `Shared location`. This option causes generated code for utilities to be placed at a shared location within the `slprj` folder in the "Code generation folder" (Simulink).
   - Select the **All Parameters** tab.

     - If it is not already cleared, clear the **Classic call interface** option.
     - If it is not already set, set the **Single output/update function** option.

   Click **Apply** and **OK**.
4. With the model open, initiate code generation and the build process for the model by using any of the following options:

- Click the **Build Model** button.
- Press **Ctrl+B**.
- Select **Code** > **C/C++ Code** > **Build Model**.
- Invoke the `rtwbuild` command from the MATLAB command line.
- Invoke the `slbuild` command from the MATLAB command line.

Some messages concerning code generation and compilation appear in the MATLAB Command Window. The initial message is:

```
### Starting build procedure for model: myAircraftExample
```

The contents of many of the succeeding messages depends on your compiler and operating system. The final messages include:

```
### Created executable slexAircraftExample.exe
### Successful  completion of build procedure for model: myAircraftExample
### Creating HTML report file myAircraftExample_codegen_rpt.html
```

The code generation folder now contains an executable, `myAircraftExample.exe` (Microsoft Windows platforms) or `myAircraftExample` (UNIX platforms). In addition, the build process has created an `slprj` folder and a `myAircraftExample_grt_rtw` folder in the "Code generation folder" (Simulink) .

Note: The build process displays a code generation report after generating the code for the `myAircraftExample` model. See "Report Generation" (Simulink Coder) for more information about how to create and use a code generation report.

---

5 To observe the contents of the working folder after the build, type the `dir` or `ls` command from the Command Window.

```
>> dir

.                   myAircraftExample.exe      myAircraftExample_grt_rtw
..                  myAircraftExample.slx      slprj
```

6 To run the executable from the Command Window, type `!slexAircraftExample`. The `!` character passes the command that follows it to the operating system, which runs the standalone `slexAircraftExample` program.

```
>> !myAircraftExample

** starting the model **
** created myAircraftExample.mat **
```

7 To see the files created in the build folder, use the `dir` or `ls` command again. The exact list of files produced varies among MATLAB platforms and versions. Here is a sample list from a Windows platform.

```
>> dir myAircraftExample_grt_rtw

.                     rt_main.obj                 myAircraftExample_data.c
..                    rtmodel.h                   myAircraftExample_data.obj
buildInfo.mat         rtw_proj.tmw                myAircraftExample_private.h
codeInfo.mat          myAircraftExample.bat       myAircraftExample_ref.rsp
defines.txt           myAircraftExample.c         myAircraftExample_types.h
html                  myAircraftExample.h
modelsources.txt      myAircraftExample.mk
rt_logging.obj        myAircraftExample.obj
```

Tip: For UNIX platforms, run the executable in the Command Window with the syntax `!./executable_name`. If preferred, run the executable from an OS shell with the syntax `./executable_name`. For more information, see "Run External Commands, Scripts, and Programs" (MATLAB).

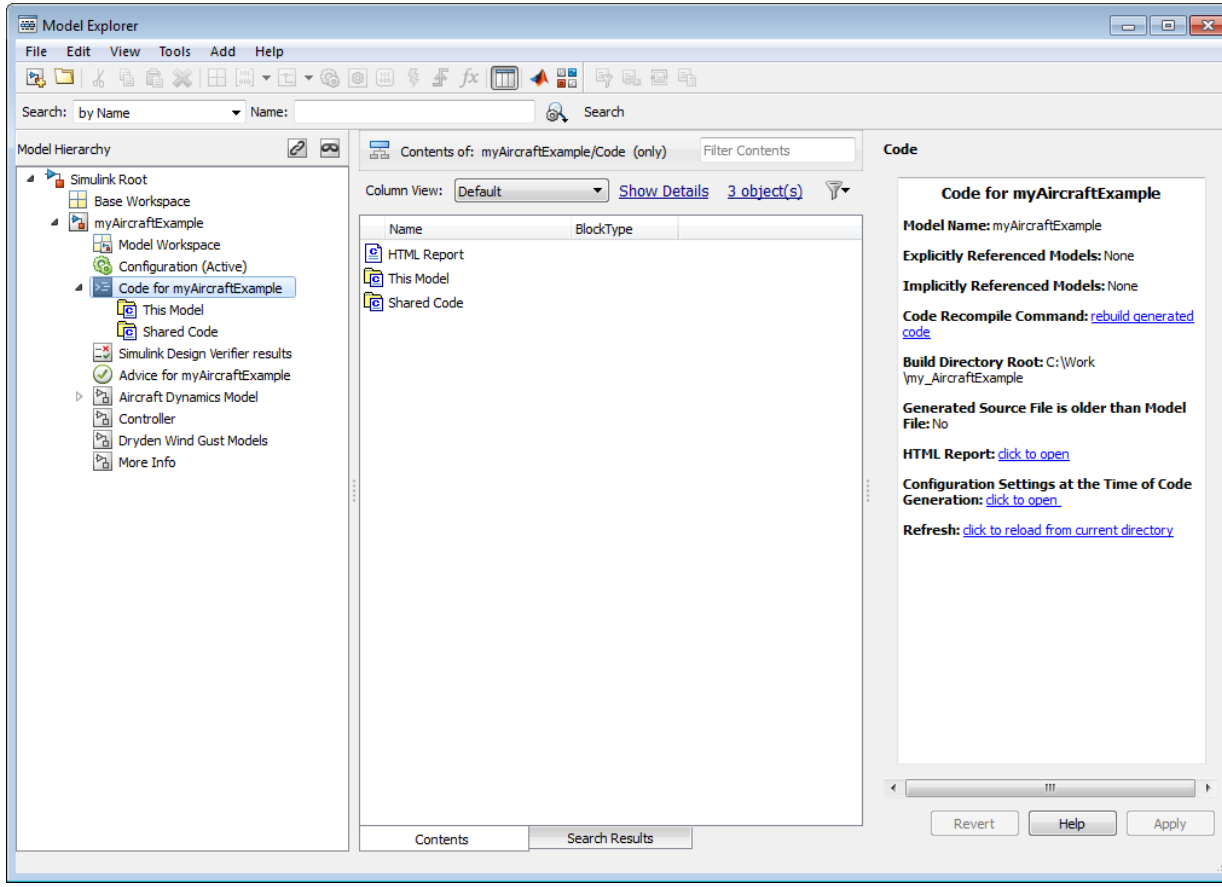
More About

- "Run External Commands, Scripts, and Programs" (MATLAB)
- "Build and Run a Program" on page 40-39
- "Rebuild a Model" on page 40-46
- "Report Generation" (Simulink Coder)

Rebuild a Model

If you update generated source code or makefiles manually to add customizations, you can rebuild the files with the `rtwrebuild` command. This command recompiles the modified files by invoking the generated makefile. Alternatively, you can use this command from the Model Explorer:

1. Open the top model. To open the Model Explorer window, select **View > Model Explorer**.
2. In the **Model Hierarchy** pane, expand the node for the model.
3. Click the **Code for *`model`*** node.
4. In the **Code** pane, next to **Code Recompile Command**, click **rebuild generated code**.



More About

- rtwrebuild (Simulink Coder)
- “Build and Run a Program” on page 40-43

Control Regeneration of Top Model Code

When you rebuild a model, by default, the build process performs checks to determine whether changes to the model or relevant settings require regeneration of the top model code. The model build regenerates top model code if any of the following conditions is true:

- The structural checksum of the model has changed.
- The top-model-only checksum has changed. The top-model-only checksum provides information about top model parameters, such as application lifespan, maximum stack size, make command, verbose and `.rtw` file debug settings, and `TLCOptions`.
- Any of the following TLC debugging model options, on the **All Parameters** tab of the Configuration Parameters dialog box, is `on`:
  - **Start TLC debugger when generating code** (`TLCDebug`)
  - **Start TLC coverage when generating code** (`TLCCoverage`)
  - **Enable TLC assertion** (`TLCAssert`)
  - **Profile TLC** (`ProfileTLC`)

Whether the top model code is regenerated, the build process calls the build process hooks and reruns the makefile. The hooks include the *`STF`*`_make_rtw_hook` functions and the post code generation command. This process recompiles and links the external dependencies.

System target file authors can perform actions related to code regeneration in the *`STF`*`_make_rtw_hook` functions that the build process calls. These actions include forcing or reacting to code regeneration. For more information, see “Control Code Regeneration Using STF_make_rtw_hook.m” on page 70-36.

Regeneration of Top Model Code

If the checks determine that top model code generation is required, the build process fully regenerates and compiles the model code. An example check is whether previously generated code is not current due to a model update.

The build process omits regeneration of the top model code when the checks indicate both:

- The top model generated code is current for the model.

- No model settings require full regeneration,

This omission can significantly reduce model build times.

With an Embedded Coder license, if you modify a code generation template (CGT) file then rebuild your model, the code generation process does not force a top model build. In this case, see "Force Regeneration of Top Model Code" on page 40-49.

Force Regeneration of Top Model Code

If you want to control or override the default top model build behavior, use one of the following command-line options:

- To ignore the checksum and force regeneration of the top model code:
  - `rtwbuild(`*`model`*`,'ForceTopModelBuild',true)`
  - `slbuild(`*`model`*`,'StandaloneRTWTarget','ForceTopModelBuild',true)`
- To clean the model build area enough to trigger regeneration of the top model code at the next build (`slbuild` only):
  ```
  slbuild(model,'CleanTopModel')
  ```

You can force regeneration of the top model code by deleting the `slprj` folder or the generated model code folder from the "Code generation folder" (Simulink) .

More About

- rtwrebuild (Simulink Coder)
- "Control Code Regeneration Using STF_make_rtw_hook.m" on page 70-36
- "Choose and Configure Build Process" on page 40-14
- "Rebuild a Model" on page 40-46
- "Reduce Build Time for Referenced Models" on page 40-50

Reduce Build Time for Referenced Models

In a parallel computing environment, you can increase the speed of code generation and compilation for models containing large model reference hierarchies. Achieve the speed by building referenced models in parallel whenever conditions allow. For example, if you have Parallel Computing Toolbox™ software, code generation and compilation for each referenced model can be distributed across the cores of a multicore host computer. If you also have MATLAB Distributed Computing Server™ software, you can distribute code generation and compilation for each referenced model across remote workers in your MATLAB Distributed Computing Server configuration.

In this section...

Parallel Building for Large Model Reference Hierarchies

The performance gain realized by using parallel builds for referenced models depends on several factors. These factors include how many models can be built in parallel for a given model referencing hierarchy, the size of the referenced models, and parallel computing resources. The resources include the number of local and/or remote workers available and the hardware attributes of the local and remote machines. Examples of hardware attributes are the amount of RAM and number of cores.

For configuration requirements that could apply to your parallel computing environment, see "Parallel Building Configuration Requirements" on page 40-51.

For a description of the general workflow for building referenced models in parallel whenever conditions allow, see "Build Models in a Parallel Computing Environment" on page 40-51.

For information on how to configure a custom embedded system target file to support parallel builds, see "Support Model Referencing" (Simulink Coder).

Note: In a MATLAB Distributed Computing Server parallel computing configuration, parallel building is designed to work interactively with the software. You can initiate

builds from the Simulink user interface or from the MATLAB Command Window using commands such as `slbuild`. You cannot initiate builds using `batch` or other batch mode workflows.

Parallel Building Configuration Requirements

The following requirements apply to configuring your parallel computing environment for building model reference hierarchies in parallel whenever conditions allow:

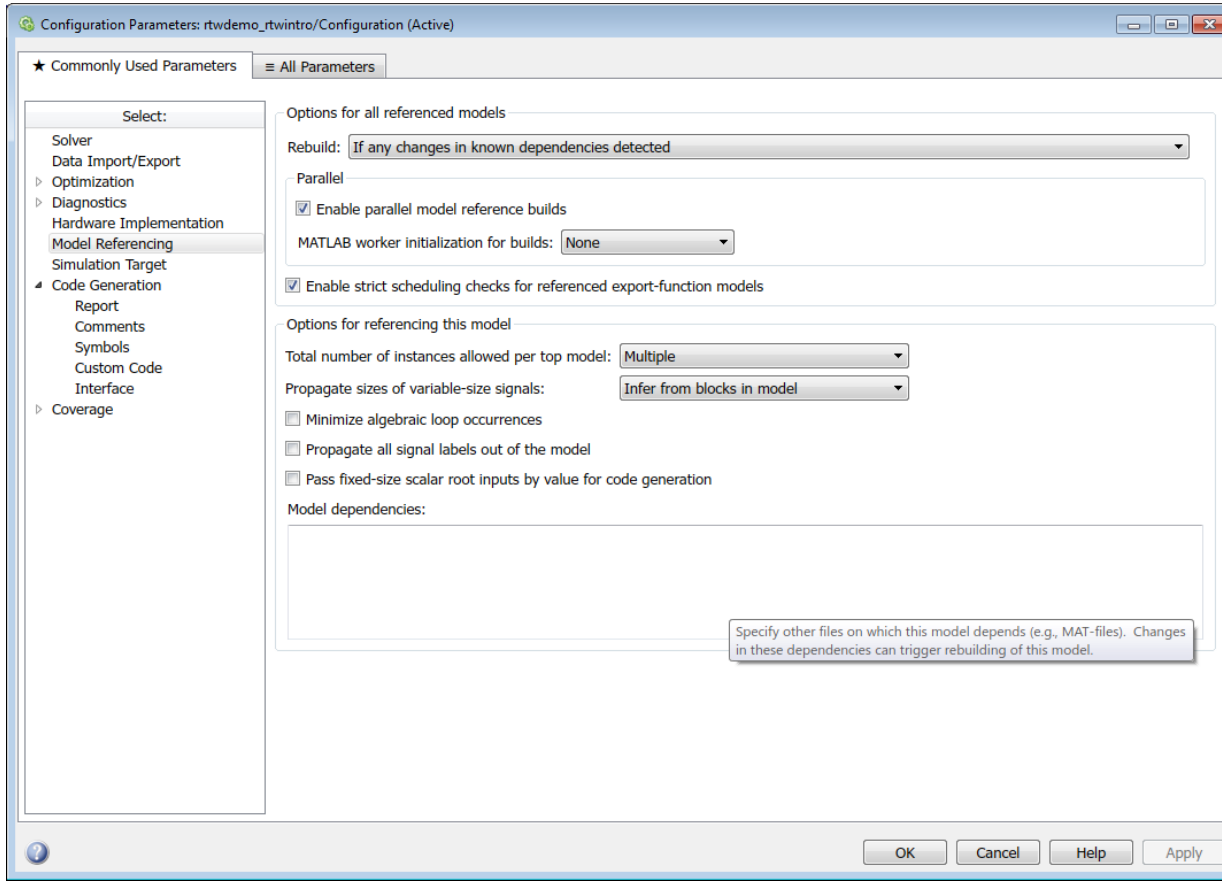
- For local pools, the host machine must have enough RAM to support the number of local workers (MATLAB sessions) that you plan to use. For example, using `parpool(4)` to create a parallel pool with four workers results in five MATLAB sessions on your machine. Each pool uses the amount of memory required for MATLAB and the code generator at startup. For memory requirements, see System Requirements for MATLAB & Simulink.
- Remote MATLAB Distributed Computing Server workers participating in a parallel build and the client machine must use a common platform and compiler.
- A consistent MATLAB environment must be set up in each MATLAB worker session as in the MATLAB client session — for example, shared base workspace variables, MATLAB path settings, and so forth. One approach is to use the `PreLoadFcn` callback of the top model. If you configure your model to load the top model with each MATLAB worker session, its preload function can be used for any MATLAB worker session setup.

Build Models in a Parallel Computing Environment

To take advantage of parallel building for a model reference hierarchy:

1. Set up a pool of local and/or remote MATLAB workers in your parallel computing environment.
   a. Make sure that Parallel Computing Toolbox software is licensed and installed.
   b. To use remote workers, make sure that MATLAB Distributed Computing Server software is licensed and installed.
   c. Issue MATLAB commands to set up the parallel pool, for example, `parpool(4)`.
2. From the top model of the model reference hierarchy, open the Configuration Parameters dialog box. Go to the **Model Referencing** pane and select the **Enable**

parallel model reference builds (Simulink) option. This selection enables the parameter **MATLAB worker initialization for builds** (Simulink).



For **MATLAB worker initialization for builds**, select one of the following values:

- `None` if it is preferable that the software does not perform special worker initialization. Specify this value if the child models in the model reference hierarchy do not rely on anything in the base workspace beyond what they explicitly set up. An example is a model load function.

- `Copy base workspace` if it is preferable that the software attempts to copy the base workspace to each worker. Specify this value if you use a setup script to prepare the base workspace for multiple models to use.
- `Load top model` if it is preferable that the software loads the top model on each worker. Specify this value if the top model in the model reference hierarchy handles the base workspace setup. An example is a model load function.

Note: Only set **Enable parallel model reference builds** for the top model of the model reference hierarchy to which it applies.

3 Optionally, turn on verbose messages for simulation builds, code generation builds, or both. If you select verbose builds, the build messages report the progress of each parallel build with the name of the model.

- To turn on verbose messages in model builds for simulation, go to **Configuration Parameters > All Parameters > Optimization** and select **Verbose accelerator builds**.
- To turn on verbose messages in model builds for code generation, go to **Configuration Parameters > All Parameters > Code Generation** and select **Verbose build**.

Verbose options control the build messages in the MATLAB Command Window and in parallel build log files.

4 Optionally, inspect the model reference hierarchy to determine, based on model dependencies, which models are built in parallel. For example, you can use the Model Dependency Viewer from the Simulink **Analysis > Model Dependencies** menu.

5 Build your model. Messages in the MATLAB Command Window record when each parallel or serial build starts and finishes. The order in which referenced models build is nondeterministic. They could build in a different order each time the model is built.

If you need more information about a parallel build, for example, if a build fails, see "Locate Parallel Build Logs" on page 40-53.

Locate Parallel Build Logs

If verbose builds are turned on when you build a model for which referenced models are built in parallel, messages in the MATLAB Command Window record when each parallel or serial build starts and finishes. For example,

```
### Initializing parallel workers for parallel model reference build.
### Parallel worker initialization complete.
### Starting parallel model reference SIM build for 'bot_model001'
### Starting parallel model reference SIM build for 'bot_model002'
### Starting parallel model reference SIM build for 'bot_model003'
### Starting parallel model reference SIM build for 'bot_model004'
### Finished parallel model reference SIM build for 'bot_model001'
### Finished parallel model reference SIM build for 'bot_model002'
### Finished parallel model reference SIM build for 'bot_model003'
### Finished parallel model reference SIM build for 'bot_model004'
### Starting parallel model reference RTW build for 'bot_model001'
### Starting parallel model reference RTW build for 'bot_model002'
### Starting parallel model reference RTW build for 'bot_model003'
### Starting parallel model reference RTW build for 'bot_model004'
### Finished parallel model reference RTW build for 'bot_model001'
### Finished parallel model reference RTW build for 'bot_model002'
### Finished parallel model reference RTW build for 'bot_model003'
### Finished parallel model reference RTW build for 'bot_model004'
```

To obtain more detailed information about a parallel build, you can examine the parallel build log. For each referenced model built in parallel, the build process generates a file named *`model`*`_buildlog.txt`, where *`model`* is the name of the referenced model. This file contains the full build log for that model.

If a parallel build completes, you can find the build log file in the build subfolder corresponding to the referenced model. For example, for a build of referenced model `bot_model004`, look for the build log file `bot_model004_buildlog.txt` in a referenced model subfolder such as *`build_folder`*`/slprj/grt/bot_model004`, *`build_folder`*`/slprj/ert/bot_model004`, or *`build_folder`*`/slprj/sim/bot_model004`. The build log (diagnostic viewer) provides a relative path to the location of each build log file.

If a parallel builds fails, you could see output similar to the following:

```
### Initializing parallel workers for parallel model reference build.
### Parallel worker initialization complete.
...
### Starting parallel model reference RTW build for 'bot_model002'
### Starting parallel model reference RTW build for 'bot_model003'
### Finished parallel model reference RTW build for 'bot_model002'
### Finished parallel model reference RTW build for 'bot_model003'
### Starting parallel model reference RTW build for 'bot_model001'
### Starting parallel model reference RTW build for 'bot_model004'
### Finished parallel model reference RTW build for 'bot_model004'
### The following error occurred during the parallel model reference RTW build for
'bot_model001':

Error(s) encountered while building model "bot_model001"

### Cleaning up parallel workers.
```

If a parallel build fails, you can find the build log file in a referenced model subfolder under the build subfolder `/par_mdl_ref/`*`model`*. For example, for a failed parallel build of model `bot_model001`, look for the build log file `bot_model001_buildlog.txt` in a subfolder such as *`build_folder`*`/par_mdl_ref/bot_model001/slprj/grt/bot_model001`, *`build_folder`*`/par_mdl_ref/bot_model001/slprj/ert/bot_model001`, or *`build_folder`*`/par_mdl_ref/bot_model001/slprj/sim/bot_model001`.

More About

- "Control Regeneration of Top Model Code" on page 40-48
- "Reduce Update Time for Referenced Models" (Simulink)
- "Build and Run a Program" on page 40-43
- "Profile Code Performance" on page 40-71
- "Support Model Referencing" on page 71-83

Relocate Code to Another Development Environment

If you require relocating the static and generated code files for a model to another development environment, use the pack-and-go utility. This condition occurs when your system or integrated development environment (IDE) does not include MATLAB and Simulink products.

In this section...

Code Relocation

The pack-n-go utility uses the tools for customizing the build process after code generation and a `packNGo` function to find and package files for building an executable image. The files are packaged in a compressed file that you can relocate and unpack using a standard `zip` utility.

To package model code files, you can do either of the following:

- Use the model option **Package code and artifacts** (Simulink Coder) in the **Configuration Parameters > Code Generation** pane. See "Package Code Using the User Interface" on page 40-56.
- Use MATLAB commands to configure a `PostCodeGenCommand` parameter with a call to the `packNGo` function. See "Package Code Using the Command-Line Interface" on page 40-58. The command-line interface provides more control over the details of code packaging.

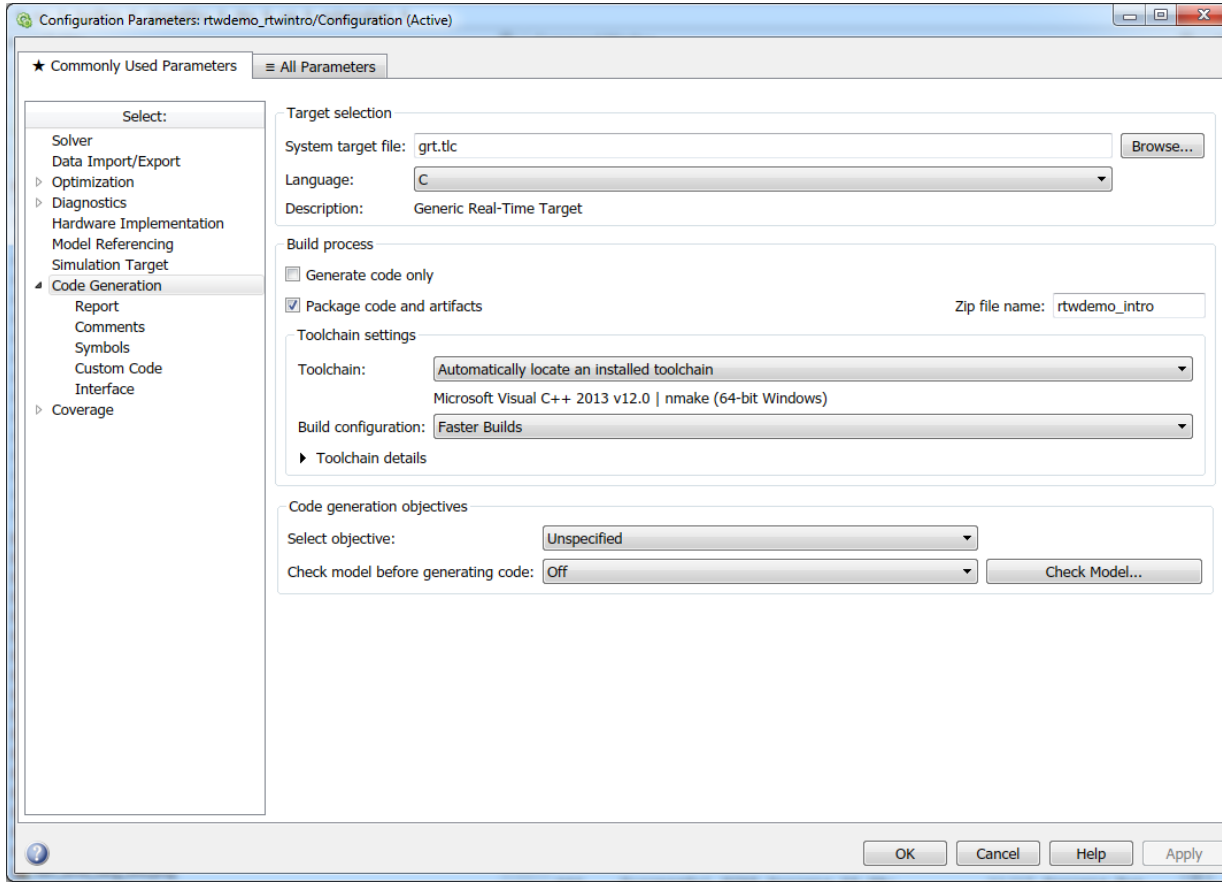
Package Code Using the User Interface

To package and relocate code for your model using the user interface:

1 Open **Configuration Parameters > Code Generation**.

2. Select the option **Package code and artifacts** (Simulink Coder). This option configures the build process to run the `packNGo` function after code generation to package generated code and artifacts for relocation.

3. In the **Zip file name** (Simulink Coder) field, enter the name of the `zip` file in which to package generated code and artifacts for relocation. You can specify the file name with or without the `.zip` extension. If you specify no extension or an extension other than `.zip`, the `zip` utility adds the `.zip` extension. If you do not specify a value, the build process uses the name *`model`*`.zip`, where *`model`* is the name of the top model for which code is being generated.



4 Apply changes and generate code for your model. To verify that it is ready for relocation, inspect the resulting `zip` file. Depending on the `zip` tool that you use, you could be able to open and inspect the file without unpacking it.

5 Relocate the `zip` file to the destination development environment and unpack the file.

Package Code Using the Command-Line Interface

To package and relocate code for your model using the command-line interface:

1 Select a structure for the `zip` file.

2 Select a name for the `zip` file.

3 Package the model code files in the `zip` file.

4 Inspect the generated `zip` file.

5 Relocate and unpack the `zip` file.

Select a Structure for the Zip File

Before you generate and package the files for a model build, decide whether you want the files to be packaged in a flat or hierarchical folder structure. By default, the `packNGo` function packages the files in a single, flat folder structure.

If...	Then Use a...
You are relocating files to an IDE that does not use the generated makefile, or the code is not dependent on the relative location of required static files	Single, flat folder structure
The destination development environment must maintain the folder structure of the source environment because it uses the generated makefile, or the code depends on the relative location of files	Hierarchical structure

If you use a hierarchical structure, the `packNGo` function creates two levels of `zip` files, a primary `zip` file, which in turn contains the following secondary `zip` files:

- `mlrFiles.zip` — files in your *`matlabroot`* folder tree

- `sDirFiles.zip` — files in and under your build folder where you initiated code generation for the model
- `otherFiles.zip` — required files not in the *`matlabroot`* or `start` folder trees

Paths for the secondary `zip` files are relative to the root folder of the primary `zip` file, maintaining the source development folder structure.

Select a Name for the Zip File

By default, the `packNGo` function names the primary `zip` file *`model`*. You have the option of specifying a different name. If you specify a file name and omit the file type extension, the function appends `.` to the name that you specify.

Package Model Code in a Zip File

Package model code files by using the `PostCodeGenCommand` configuration parameter, `packNGo` function, and build information object for the model. You can set up the packaging operation to use:

- A system generated build information object.

  In this case, before generating the model code, use `set_param` to set the configuration parameter `PostCodeGenCommand` to an explicit call to the `packNGo` function. For example:

  ```
  set_param(bdroot, 'PostCodeGenCommand', 'packNGo(buildInfo);');
  ```

  After generating and writing the model code to disk and before generating a makefile, this command instructs the build process to evaluate the call to `packNGo`. This command uses the system generated build information object for the currently selected model.

- A build information object that you construct programmatically.

  In this case, you could use other build information functions to include paths and files selectively in the build information object that you then specify with the `packNGo` function. For example:

  ```
  .
  .
  .
  myModelBuildInfo = RTW.BuildInfo;
  addSourceFiles(myModelBuildInfo, {'test1.c' 'test2.c' 'driver.c'});
  .
  ```

```
.
.
packNGo(myModelBuildInfo);
```

The following examples show how you can change the default behavior of `packNGo`.

To...	Specify...
Change the structure of the file packaging to hierarchical	`packNGo(buildInfo, {'packType' 'hierarchical'});`
Rename the primary `zip` file	`packNGo(buildInfo, {'fileName' 'zippedsrcs'});`
Change the structure of the file packaging to hierarchical and rename the primary `zip` file	`packNGo(buildInfo, {'packType' 'hierarchical'... 'fileName' 'zippedsrcs'});`
Include header files found on the include path in the `zip` file	`packNGo(buildInfo, {'minimalHeaders' false});`
Generate warnings for parse errors and missing files	`packNGo(buildInfo, {'ignoreParseError' true... 'ignoreFileMissing' true});`

Note: The `packNGo` function can potentially modify the build information passed in the first `packNGo` argument. As part of packaging model code, `packNGo` could find additional files from source and include paths recorded in build information for the model and add them to the build information.

Inspect a Generated Zip File

To verify that it is ready for relocation, inspect the generated `zip` file. Depending on the `zip` tool that you use, you could be able to open and inspect the file without unpacking it. If unpacking the file and you packaged the model code files as a hierarchical structure, unpacking requires you to unpack the primary and secondary `zip` files. When you unpack the secondary `zip` files, relative paths of the files are preserved.

Relocate and Unpack a Zip File

Relocate the generated `zip` file to the destination development environment and unpack the file.

Code Packaging Example

This example shows how to package code files generated for the example model `rtwdemo_rtwintro` using the command-line interface:

1. Set your working folder to a writable folder.
2. Open the model `rtwdemo_rtwintro` and save a copy to your working folder.
3. Enter the following MATLAB command:

   ```
   set_param('rtwdemo_rtwintro', 'PostCodeGenCommand',...
   'packNGo(buildInfo, {''packType'' ''hierarchical''})');
   ```

   You must double the single-quotes due to the nesting of character arrays `'packType'` and `'hierarchical'` within the character array that specifies the call to `packNGo`.
4. Generate code for the model.
5. Inspect the generated `zip` file, `rtwdemo_rtwintro.zip`. The `zip` file contains the two secondary `zip` files, `mlrFiles.zip` and `sDirFiles.zip`.
6. Inspect the `zip` files `mlrFiles.zip` and `sDirFiles.zip`.
7. Relocate the `zip` file to a destination environment and unpack it.

Build Integrated Code Outside the Simulink Environment

Identify required files and interfaces for calling generated code in an external build process.

Learn how to:

- Collect files required for building integrated code outside of Simulink®.
- Interface with external variables and functions.

For information about the example model and related examples, see “Generate C Code from a Control Algorithm for an Embedded System”.

Collect and Build Required Data and Files

The code that Embedded Coder® generates requires support files that MathWorks® provides. To relocate the generated code to another development environment, such as a dedicated build system, you must relocate these support files. You can package these files in a zip file by using the `packNGo` utility. This utility finds and packages the files

that you need to build an executable image. The utility uses tools for customizing the build process after code generation, which include a `buildinfo_data` structure, and a `packNGo` function. These files include external files that you identify in the **Code Generation > Custom Code** pane in the Model Configuration Parameters dialog box. The utility saves the `buildinfo` MAT-file in the *`model`*`_ert_rtw` folder.

Open the example model, rtwdemo_PCG_Eval_P5.

This model is configured to run `packNGo` after code generation.

Generate code from the entire model.

To generate the zip file manually:

1. Load the file `buildInfo.mat` (located in the `rtwdemo_PCG_Eval_P5_ert_rtw` subfolder).
2. At the command prompt, enter the command `packNGo(buildInfo)`.

The number of files in the zip file depends on the version of Embedded Coder® and on the configuration of the model that you use. The compiler does not require all of the files in the zip file. The compiled executable size (RAM/ROM) depends on the linking process. The linker likely includes only the object files that are necessary.

Integrating the Generated Code into an Existing System

This example shows how to integrate the generated code into an existing code base. The example uses the Eclipse™ IDE and the Cygwin™/gcc compiler. The required integration tasks are common to all integration environments.

Overview of Integration Environment

A full embedded controls system consists of multiple hardware and software components. Control algorithms are just one type of component. Other components can be:

- An operating system (OS)
- A scheduling layer
- Physical hardware I/O
- Low-level hardware device drivers

Typically, you do not use the generated code in these components. Instead, the generated code includes interfaces that connect with these components. MathWorks® provides

hardware interface block libraries for many common embedded controllers. For examples, see the Embedded Targets block library.

This example provides files to show how you can build a full system. The main file is `example_main.c`, which contains a simple main function that performs only basic actions to exercise the code.

View example_main.c.

```
int_T main(void)
{
  /* Initialize model */
  rt_Pos_Command_Arbitration_Init(); /* Set up the data structures for chart*/
  rtwdemo_PCG__Define_Throt_Param(); /* SubSystem: '<Root>/Define_Throt_Param' */
  defineImportData();                /* Defines the memory and values of inputs */

  do /* This is the "Schedule" loop.
      Functions would be called based on a scheduling algorithm */
  {
    /* HARDWARE I/O */

    /* Call control algorithms */
    PI_Cntrl_Reusable((*pos_rqst),fbk_1,&rtwdemo_PCG_Eval_P5_B.PI_ctrl_1,
                      &rtwdemo_PCG_Eval_P5_DWork.PI_ctrl_1);
    PI_Cntrl_Reusable((*pos_rqst),fbk_2,&rtwdemo_PCG_Eval_P5_B.PI_ctrl_2,
                      &rtwdemo_PCG_Eval_P5_DWork.PI_ctrl_2);
    pos_cmd_one = rtwdemo_PCG_Eval_P5_B.PI_ctrl_1.Saturation1;
    pos_cmd_two = rtwdemo_PCG_Eval_P5_B.PI_ctrl_2.Saturation1;

    rtwdemo_Pos_Command_Arbitration(pos_cmd_one, &Throt_Param, pos_cmd_two,
                                    &rtwdemo_PCG_Eval_P5_B.sf_Pos_Command_Arbitration);
```

The file:

- Defines function interfaces (function prototypes).
- Includes files that declare external data.
- Defines `extern` data.
- Initializes data.
- Calls simulated hardware.
- Calls algorithmic functions.

The order of function execution matches the order of subsystem execution in the test harness model and in `rtwdemo_PCG_Eval_P5.h`. If you change the order of execution in `example_main.c`, results that the executable image produces differ from simulation results.

Match System Interfaces

Integration requires matching the *Data* and *Function* interfaces of the generated code and the existing system code. In this example, the `example_main.c` file imports and exports the data through `#include` statements and `extern` declarations. The file also calls the functions from the generated code.

Connect Input Data

The system has three input signals: `pos_rqst`, `fbk_1`, and `fbk_2`. The generated code accesses the two feedback signals through direct reference to imported global variables (storage class `ImportedExtern`). The code accesses the position signal through an imported pointer (storage class `ImportedExternPointer`).

The handwritten file `defineImportedData.c` defines the variables and the pointer. The generated code does not define the variables and the pointer because the handwritten code defines them. Instead, the generated code declares the imported data (`extern`) in the file `rtwdemo_PCG_Eval_P5_Private.h`. In a real system, the data typically comes from other software components or from hardware devices.

View defineImportedData.c.

```
/* Define imported data */
#include "rtwtypes.h"
#include "defineImportedData.h"

real_T fbk_1;
real_T fbk_2;
real_T dummy_pos_value = 10.0;
real_T *pos_rqst;
void defineImportData(void)
{
    pos_rqst = &dummy_pos_value;
}
```

View rtwdemo_PCG_Eval_P5_Private.h.

```
/* Imported (extern) block signals */
extern real_T fbk_1;                   /* '<Root>/fbk_1' */
extern real_T fbk_2;                   /* '<Root>/fbk_2' */

/* Imported (extern) pointer block signals */
extern real_T *pos_rqst;               /* '<Root>/pos_rqst' */
```

Connect Output Data

In this example, you do not access the output data of the system. The example "Test Generated Code" shows how you can save the output data to a standard log file. You can access the output data by referring to the file `rtwdemo_PCG_Eval_P5.h`.

View rtwdemo_PCG_Eval_P5.h.

Access Additional Data

The generated code contains several structures that store commonly used data including:

- Block state values (integrator, transfer functions)
- Local parameters
- Time

The table lists the common data structures. Depending on the configuration of the model, some or all of these structures appear in the generated code. The data is declared in the file `rtwdemo_PCG_Eval_P5.h`, but in this example, you do not access this data.

```
Data Type      Data Name     Data Purpose

Constants      |model_cP|    Constant parameters
Constants      |model_cB|    Constant block I/O
Output         |model_U|     Root and atomic subsystem input
Output         |model_Y|     Root and atomic subsystem output
Internal data  |model_B|     Value of block output
Internal data  |model_D|     State information vectors
Internal data  |model_M|     Time and other system level data
Internal data  |model_Zero|  Zero-crossings
Parameters     |model_P|     Parameters
```

Match Function Call Interfaces

By default, functions that the code generator generates have a `void Func(void)` interface. If you configure the model or atomic subsystem to generate reentrant code, the code generator creates a more complex function prototype. In this example, the `example_main` function calls the generated functions with the correct input arguments.

```
PI_Cntrl_Reusable((*pos_rqst),fbk_1,&rtwdemo_PCG_Eval_P5_B.PI_ctrl_1,
                  &rtwdemo_PCG_Eval_P5_DWork.PI_ctrl_1);
PI_Cntrl_Reusable((*pos_rqst),fbk_2,&rtwdemo_PCG_Eval_P5_B.PI_ctrl_2,
                  &rtwdemo_PCG_Eval_P5_DWork.PI_ctrl_2);
pos_cmd_one = rtwdemo_PCG_Eval_P5_B.PI_ctrl_1.Saturation1;
pos_cmd_two = rtwdemo_PCG_Eval_P5_B.PI_ctrl_2.Saturation1;

rtwdemo_Pos_Command_Arbitration(pos_cmd_one, &Throt_Param, pos_cmd_two,
                                &rtwdemo_PCG_Eval_P5_B.sf_Pos_Command_Arbitration);
```

Calls to the function `PI_Cntrl_Reusable` use a mixture of separate, unstructured global variables and Simulink® Coder™ data structures. The handwritten code defines these variables. The structure types are defined in `rtwdemo_PCG_Eval_P5.h`.

Build Project in Eclipse™ Environment

This example uses the Eclipse™ IDE and the Cygwin™ GCC debugger to build the embedded system. The example provides installation files for both programs. Software components and versions numbers are:

- Eclipse™ SDK 3.2
- Eclipse™ CDT 3.3
- Cygwin™/GCC 3.4.4-1
- Cygwin™/GDB 20060706-2

To install and use Eclipse™ and GCC, see “Install and Use Cygwin and Eclipse”.

You can install the files for this example by clicking this hyperlink:

Set up the build folder.

Alternatively, to install the files manually:

1 Create a build folder (`Eclipse_Build_P5`).

2. Unzip the file `rtwdemo_PCG_Eval_P5.zip` into the build folder.
3. Delete the files `rtwdemo_PCG_Eval_P5.c`, `ert_main.c` and `rt_logging.c`, which are replaced by `example_main.c`.

You can use the Eclipse™ debugger to step through and evaluate the execution behavior of the generated C code. See the example "Install and Use Cygwin and Eclipse".

To exercise the model with input data, see "Test Generated Code".

Related Topics

- "Generate Component Source Code for Export to External Code Base"
- "Generate Shared Library for Export to External Code Base"

packNGo Function Limitations

The following limitations apply to the `packNGo` function:

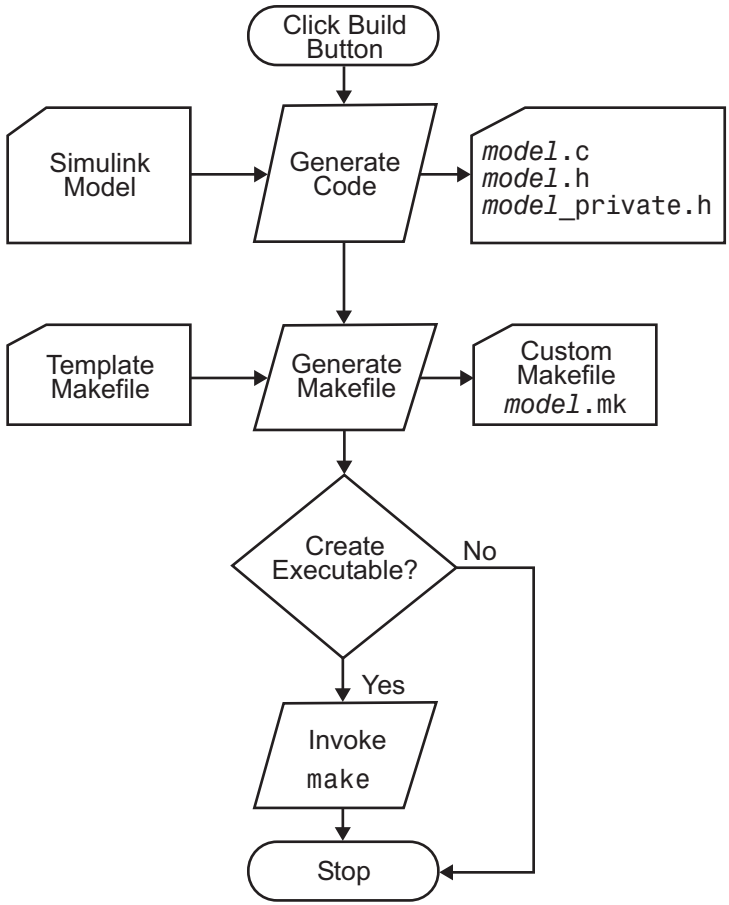
- The function operates on source files, such as `*.c`, `*.cpp`, and `*.h` files, only. The function does not support compile flags, defines, or makefiles.
- Unnecessary files could be included. The function could find additional files from source and include paths recorded in build information for the model and include them, even if they are not used.

More About

- packNGo (Simulink Coder)
- "Choose and Configure Build Process" on page 40-14
- "Build and Run a Program" on page 40-43
- "Executable Program Generation" on page 40-68

Executable Program Generation

The following figure shows how the process for program building.

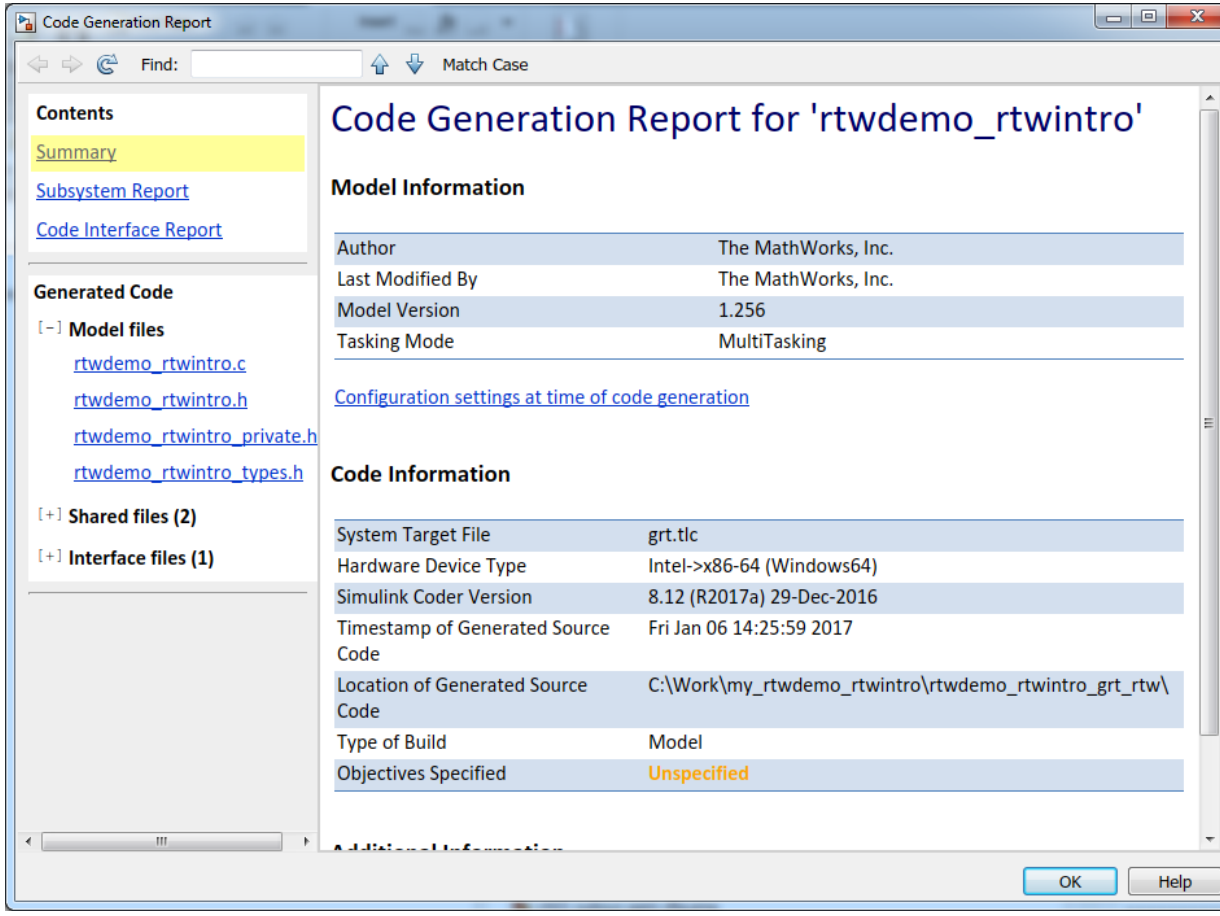


During the final stage of processing, the build process invokes the generated makefile, *model*`.mk`, which in turn compiles and links the generated code. On PC platforms, a batch file is created to invoke the generated makefile. The batch file sets up the environment for invoking the `make` utility and related compiler tools. To avoid recompiling C files, the `make` utility performs date checking on the dependencies between the object and C files; only out-of-date source files are compiled. Optionally, the makefile can download the resulting executable image to your target hardware.

This stage is optional, as illustrated by the control logic in the preceding figure. You could choose to omit this stage (for example, if you are targeting an embedded microcontroller board).

To omit this stage of processing, select the **Configuration Parameters > Code Generation** pane and select the **Generate code only** check box. You can then cross-compile your code and download it to your target hardware.

If you select the **Configuration Parameters > Code Generation > Report** pane and select **Create code generation report**, the code generator produces a navigable summary of source files when the model is built. The report files occupy a folder called `html` within the build folder. The following display shows an example of an HTML code generation report for a generic real-time (GRT) system target file.



More About

- “Choose and Configure Build Process” on page 40-14
- “Build and Run a Program” on page 40-43
- “Executable Program Generation” on page 40-68

Profile Code Performance

By profiling the performance of generated code, you can help verify that the code meets performance requirements.

Profiling can be especially important early in the development cycle for identifying potential architectural issues that can be more expensive to address later in the process. Profiling can also identify bottlenecks and procedural issues that indicate a need for optimization, for example, with an inner loop or inline code.

Note: If you have an Embedded Coder license, see "Code Execution Profiling" for an alternative and simpler approach based on software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations.

In this section...

Use the Profile Hook Function Interface

You can profile code generated with code generation technology by using a Target Language Compiler (TLC) hook function interface.

To use the profile hook function interface:

1. For your system target file, create a TLC file that defines the following hook functions. Write the functions so that they specify profiling code. The code generator adds the hook function code to code generated for atomic systems in the model.

Function	Input Arguments	Output Type	Description
`ProfilerHeaders`	`void`	Array of header file names	Return an array of the header file names to be included in the generated code.
`ProfilerTypedefs`	`void`	`typedefs`	Generate code statements for profiler type definitions.

Function	Input Arguments	Output Type	Description
`ProfilerGlobal-Data`	`system`	Global data for the specified `system`	Generate code statements that declare global data.
`ProfilerExtern-DataDecls`	`system`	`extern` declarations for the specified `system`	Generate code statements that create global `extern` declarations.
`ProfilerSystem-Decls`	`system`, `functionType`	Declarations for the specified `system` for the specified `functionType`	Generate code for required variable declarations within the scope of an atomic subsystem `Output`, `Update`, `OutputUpdate`, or `Derivatives` function.
`ProfilerSystem-Start`	`system`, `functionType`	Profiler start commands for the specified `system` and `functionType`	Generate code that starts the profiler within the scope of an atomic subsystem `Output`, `Update`, `OutputUpdate`, or `Derivatives` function.
`ProfilerSystem-Finish`	`system`, `functionType`	Profiler end commands for the specified `system` and `functionType`	Generate code that stops the profiler within the scope of the `Output`, `Update`, `OutputUpdate`, or `Derivatives` functions of an atomic subsystem.
`ProfilerSystem-Terminate`	`system`	Profiler termination code for the specified `system`	Generate code that terminates profiling (and possibly reports results) for an atomic subsystem.

For an example TLC file, see *`matlabroot`*`/toolbox/rtw/rtwdemos/rtwdemo_profile_hook.tlc`.

2 In your *`target`*`.tlc` file, define the following global variables.

Define...	To Be...
`ProfileGenCode`	`TLC_TRUE` or 1 to turn on profiling (`TLC_FALSE` or 0 to turn off profiling)
`ProfilerTLC`	The name of the TLC file that you created in step 1

A quick way to define global variables is to define the parameters with the `-a` option. You can apply this option by using the `set_param` command to set the model configuration parameter `TLCOptions`. For example,

```
>> set_param(gcs,'TLCOptions','-aProfileGenCode=1 -aProfilerTLC="rtwdemo_profile_hook.tlc"')
```

3. Consider setting configuration parameters for generating a code generation report. You can then examine the profiling code in the context of the code generated for the model.
4. Build the model. The build process embeds the profiling code in the hook function locations in the generated code for the model.
5. Run the generated executable file. In the MATLAB Command Window, enter `!`*`model-name`*. You see the profiling report you programmed in the profiling TLC file that you created. For example, a profile report could list the number of calls made to each system in a model and the number of CPU cycles spent in each system.

For details on programming a `.tlc` file and defining TLC configuration variables, see "Target Language Compiler" (Simulink Coder).

Profile Hook Function Interface Limitation

The TLC hook function interface for profiling code performance does not support the S-function system target file (`rtwsfcn.tlc`).

More About

- "Build and Run a Program" on page 40-43
- "Code Execution Profiling"

41

Host/Target Communication in Simulink Coder

Set Up and Use Host/Target Communication Channel

External mode allows two separate systems, a *host* and a *target*, to communicate. The host is the computer where the MATLAB and Simulink environments execute. The target is the computer where the executable created by the code generation and build process runs.

The host (the Simulink environment) transmits messages requesting the target to accept parameter changes or to upload signal data. The target responds by executing the request. External mode communication is based on a *client/server* architecture, in which the Simulink environment is the client and the target is the server.

In this section...

What You Can Do with a Host/Target Communication Channel

You can use a host/target communication channel to:

- Modify, or *tune*, block parameters in real time. In external mode, whenever you change parameters in the block diagram, the Simulink engine downloads them to the executing target program. You can tune your program parameters without recompiling.
- View and log block outputs in many types of blocks and subsystems. You can monitor and store signal data from the executing target program, without writing special interface code. You can define the conditions under which data is uploaded from target to host. For example, data uploading can be triggered by a selected signal crossing zero in a positive direction. Alternatively, you can manually trigger data uploading.

External mode establishes a communication channel between the Simulink engine and generated code. The channel's low-level *transport layer* handles the physical transmission of messages. The Simulink engine and the generated model code are independent of this layer. The transport layer and the code directly interfacing to it are isolated in separate modules that format, transmit, and receive messages and data packets.

This design allows for different targets to use different transport layers. For example:

- ERT, GRT, and RSim targets support external mode host/target communication by using TCP/IP and serial (RS-232) communication.
- The Simulink Real-Time product uses a customized transport layer.
- The Simulink Desktop Real-Time product uses shared memory communication.
- Target hardware platforms supported by Simulink use serial or TCP/IP communication.

For an example of using external mode, see "Set Up an External Mode Communication Channel" on page 41-3.

Set Up an External Mode Communication Channel

External Mode Communication Channel Setup

External mode is a very useful environment for rapid prototyping. This example consists of four parts, each of which depends on completion of the preceding ones, in order. The four parts correspond to the steps that you follow in simulating, building, running, and tuning an actual real-time application.

1. Set up the model.
2. Build the target executable.
3. Run the external mode target program.
4. Tune parameters.

The example uses the GRT target. It does not require hardware other than the computer on which you run the Simulink and Simulink Coder software. The generated executable in this example runs on the host computer in a separate process from MATLAB and Simulink.

Set Up the Model

In this part of the example, you create a simple model, `ex_extModeExample`. You also create a folder called `ext_mode_example` to store the model and the generated executable.

To create the folder and the model:

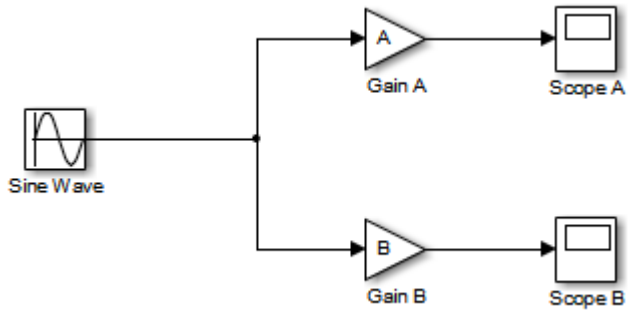
1. From the MATLAB command line, type:

   ```
   mkdir ext_mode_example
   ```

2. Make `ext_mode_example` your working folder:

   ```
   cd ext_mode_example
   ```

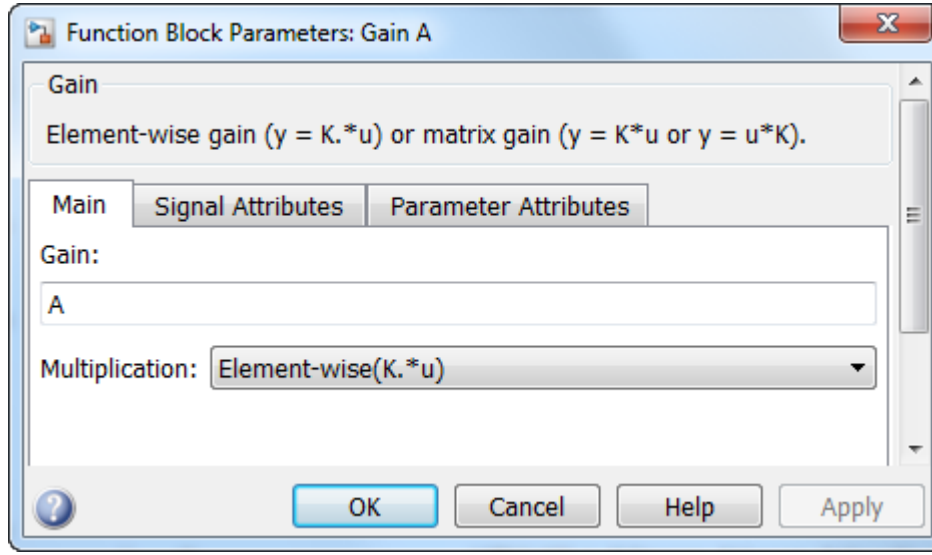
3. Create a model in Simulink with a Sine Wave block for the input signal, two Gain blocks in parallel, and two Scope blocks. Be sure to label the Gain and Scope blocks as shown.



4. Define and assign two MATLAB workspace variables, `A` and `B`:

   ```
   A = 2;
   B = 3;
   ```

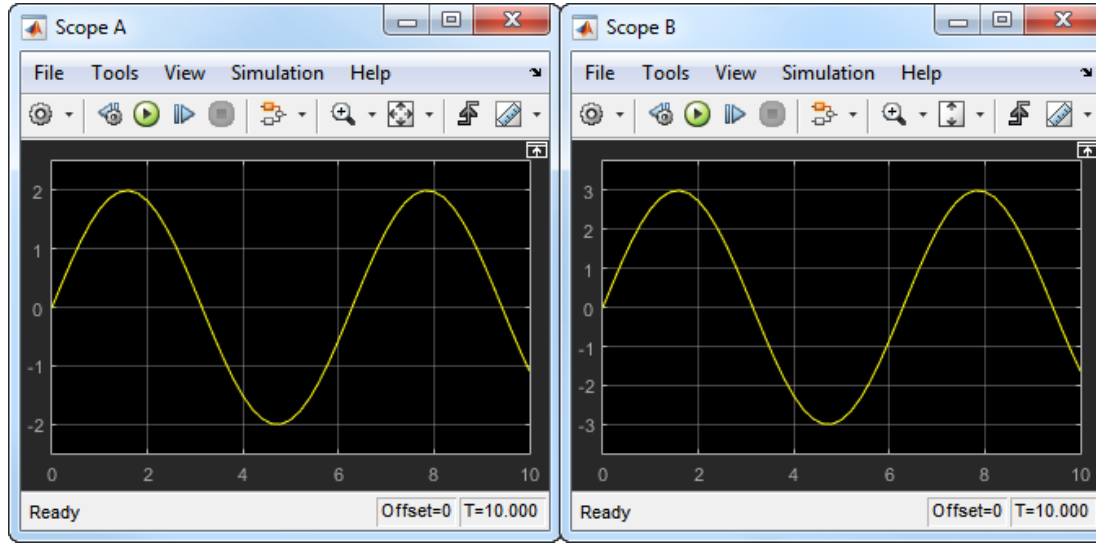
5. Open Gain block A and set its **Gain** parameter to the variable `A`.



6 Open Gain block B and set its **Gain** parameter to the variable `B`.

When the target program is built and connected to Simulink in external mode, you can download new gain values to the executing target program. To do this, you can assign new values to workspace variables `A` and `B`, or edit the values in the block parameters dialog box. For more information, see "Tune Parameters" on page 41-12.

7 Verify operation of the model. Open the Scope blocks and run the model. When `A = 2` and `B = 3`, the output appears as shown.



8 Save the model as `ex_extModeExample`.

Build the Target Executable

Set up the model and code generation parameters required for an external mode compatible target program. Then, generate code and build the target executable.

1 Open the Configuration Parameters dialog box by selecting **Simulation** > **Model Configuration Parameters**.

2 Select the **Solver** pane.

3 In the **Solver options** subpane:

   a In the **Type** field, select `Fixed-step`.

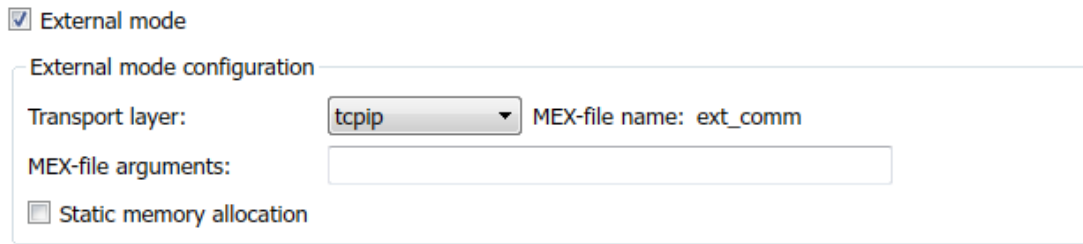
   b In the **Solver** field, select `discrete (no continuous states)`.

   c Open **Additional options**. In the **Fixed-step size** field, specify `0.1`. (Otherwise, when you generate code, the Simulink Coder build process posts a warning and supplies a value.)

   Leave **Start time** set to the default value of `0.0`.

   Click **Apply**.

4. Select the **Data Import/Export** pane, and clear the **Time** and **Output** check boxes. In this example, data is not logged to the workspace or to a MAT-file. Click **Apply**.
5. Select the **Optimization** > **Signals and Parameters** pane. Make sure that **Default parameter behavior** is set to `Tunable`. Inlined parameters are not part of this example. If you have made changes, click **Apply**.
6. Select the **Code Generation** pane. By default, the generic real-time (GRT) target is selected.
7. Select the **Code Generation** > **Interface** pane. In the **Data exchange interface** section, select **External mode**. This selection enables generation of external mode support code and displays additional external mode configuration parameters.
8. In the **External mode configuration** section, make sure that the default value `tcpip` is selected for the **Transport layer** parameter.



External mode supports communication via TCP/IP, serial, and custom transport protocols. The **MEX-file name** field specifies the name of a MEX-file that implements host and target communication on the host side. The default for TCP/IP is `ext_comm`, a MEX-file provided with the Simulink Coder software. You can override this default by supplying other files. If you need to support other transport layers, see "Create a Transport Layer for External Communication" (Simulink Coder).

The **MEX-file arguments** field lets you specify arguments, such as a TCP/IP server port number, to be passed to the external interface program. These arguments are specific to the external interface that you are using. For information on setting these arguments, see "MEX-File Optional Arguments for TCP/IP Transport" (Simulink Coder) and "MEX-File Optional Arguments for Serial Transport" (Simulink Coder).

This example uses the default arguments. Leave the **MEX-file arguments** field blank.

9. Click **Apply** to save the external mode settings.

10 Save the model.

11 Select the **Code Generation** pane. Make sure that **Generate code only** is cleared, and then, in the model window, press **Ctrl+B** to generate code and create the target program. The software creates the `ex_extModeExample` target executable in your working folder.

Run the External Mode Target Program

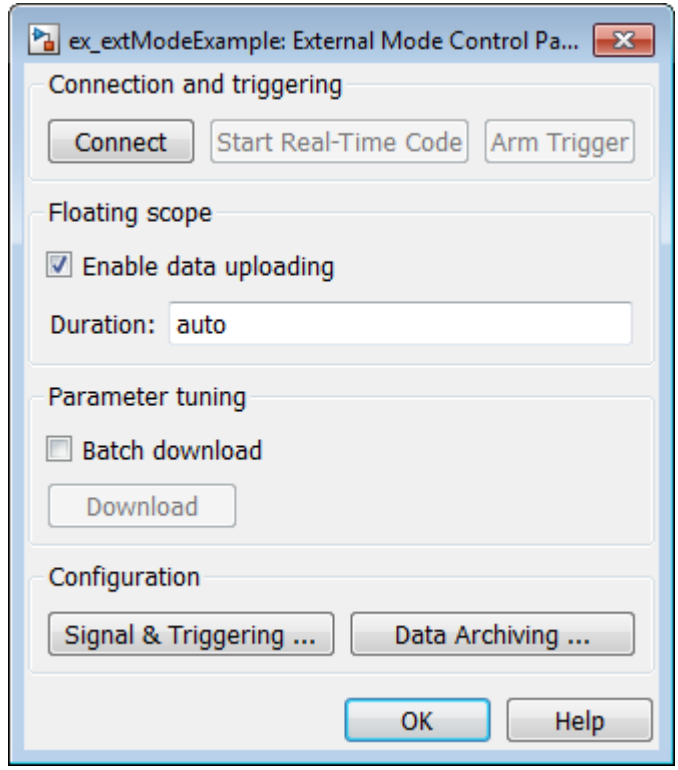
You now run the `ex_extModeExample` target executable and use Simulink as an interactive front end to the running target program. The executable file is in your working folder. Run the target program and establish communication between Simulink and the target.

Note: An external mode program like `ex_extModeExample` is a host-based executable. Its execution is not tied to a real-time operating system (RTOS) or a periodic timer interrupt, and it does not run in real time. The program just runs as fast as possible, and the time units it counts off are simulated time units that do not correspond to time in the world outside the program.

The External Signal & Triggering dialog box (accessed from the External Mode Control Panel) displays a list of blocks in your model that support external mode signal monitoring and logging. In the dialog box, you can configure the signals that are viewed, how they are acquired, and how they are displayed.

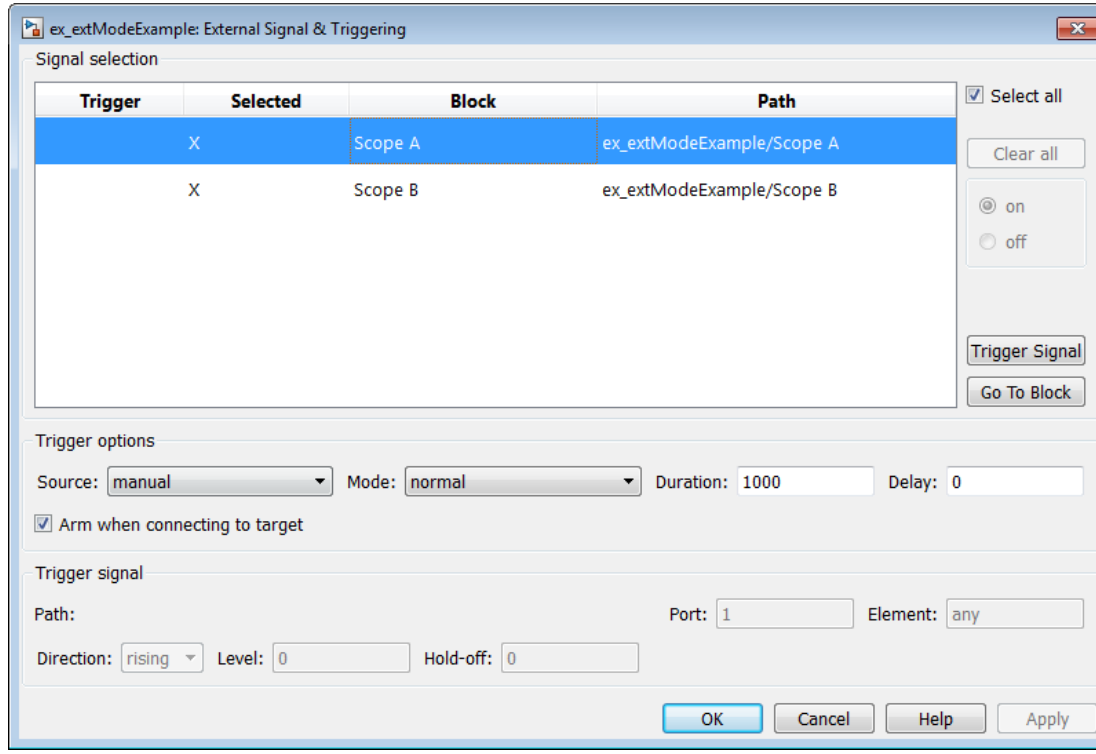
In this example, you observe and use the default settings of the External Signal & Triggering dialog box.

1 From the **Code** menu of the model diagram, select **External Mode Control Panel**. This control panel is where you configure signal monitoring and data archiving. You can also connect to the target program, and start and stop execution of the model code.



- After the target program starts, you use the top row of buttons.
- The **Signal & Triggering** button opens the External Signal & Triggering dialog box. Use this dialog box to select the signals that are collected from the target system and viewed in external mode. You can also select a signal that triggers uploading of data when certain signal conditions are met, and define the triggering conditions.
- The **Data Archiving** button opens the Enable Data Archiving dialog box. Use data archiving to save data sets generated by the target program for future analysis. This example does not use data archiving. See "Configure Host Archiving of Target Program Signal Data" on page 41-31 .

2 Click the **Signal & Triggering** button to open the External Signal & Triggering dialog box. The default configuration selects all signals for monitoring and sets signal monitoring to begin once the host and target programs are connected.



3 Make sure that the External Signal & Triggering dialog box options are set to these defaults:

- **Select all** check box is selected. Signals in the **Signal selection** list are marked with an `X` in the **Selected** column.
- Under **Trigger options:**
  - **Source**: `manual`
  - **Mode**: `normal`
  - **Duration**: `1000`
  - **Delay**: `0`
  - **Arm when connecting to target**: selected

To close the External Signal & Triggering dialog box, click **OK**. Then, close the External Mode Control Panel.

For descriptions of the External Signal & Triggering dialog box parameters, see "Configure Host Monitoring of Target Program Signal Data" on page 41-23.

4 To run the target program, open an operating system command window (on UNIX systems, a terminal emulator window). At the command prompt, use `cd` to navigate to the `ext_mode_example` folder to which you generated the target program executable.

Enter this command:

```
ex_extModeExample -tf inf -w
```

Note Alternatively, you can run the target program from the MATLAB Command Window, using the following syntax.

```
!ex_extModeExample -tf inf -w &
```

The target program begins execution, and enters a wait state.

The `-tf` switch overrides the stop time set for the model in Simulink. The `inf` value directs the model to run indefinitely. The model code runs until the target program receives a stop message from Simulink.

The `-w` switch instructs the target program to enter a wait state until it receives a **Start Real-Time Code** message from the host. If you want to view data from time step 0 of the target program execution, or if you want to modify parameters before the target program begins execution of model code, this switch is required.

5 Open the Scope blocks in the model. Signals are not visible on the scopes. When you connect Simulink to the target program and begin model execution, the signals generated by the target program become visible on the scope displays.

6 Before communication between the model and the target program can begin, the model must be in external mode. To enable external mode, from the **Simulation** > **Mode** menu, select **External**.

7 Reopen the External Mode Control Panel (found in the **Code** menu) and click **Connect**. This action initiates a handshake between Simulink and the target program. When Simulink and the target are connected, the **Start Real-Time**

Code button becomes enabled, and the label of the **Connect** button changes to **Disconnect**.

8 Click **Start Real-Time Code**. The outputs of Gain blocks A and B are displayed on the two scopes in your model.

You have established communication between Simulink and the running target program. You can now tune block parameters in Simulink and observe the effects the parameter changes have on the target program.

Tune Parameters

You can change the gain factor of either Gain block by assigning a new value to the variable `A` or `B` in the MATLAB workspace. When you change block parameter values in the workspace during a simulation, you must explicitly update the block diagram with these changes. When you update the block diagram, the new values are downloaded to the target program.

Under certain conditions, you can also tune the expressions that you use to specify block parameter values. To change an expression during simulation, open the block dialog box.

1 At the command prompt, assign new values to both variables, for example:

```
A = 0.5;
B = 3.5;
```

2 Open the `ex_extModeExample` model window. From the **Simulation** menu, select **Update Diagram**. As soon as Simulink has updated the block parameters, the new gain values are downloaded to the target program, and the scopes are changed because of the gain change.

3 In the Sine Wave block dialog box, set **Amplitude** to `0.5`. Click **Apply** or **OK**.

When you click **Apply** or **OK**, the simulation downloads the new block parameter value to the target program. The Scope block displays the change to reflect the new amplitude value.

4 To simultaneously disconnect host/target communication and end execution of the target program, from the **Simulation** menu, select **Stop**. Alternatively, in the External Mode Control Panel, click **Stop Real-Time Code**.

You cannot change the sample time of the Sine Wave block during simulation. Block sample times are part of the structural definition of the model and are part of the generated code. Therefore, if you want to change a block sample time, you must stop the external mode simulation, reset the sample time of the block, and rebuild the executable.

Block parameter tunability during external mode simulation depends on the way that the generated code represents block parameters.

For example, in the Gain A block dialog box, you cannot change the expression `A` in the **Gain** parameter during simulation. Instead, you must change the value of the variable `A` in the base workspace. You cannot change the expression because the generated code does not allocate storage in memory for the **Gain** parameter. Instead, the code creates a field `A` in a structure:

```
/* Parameters (auto storage) */
struct P_ex_extModeExample_T_ {
  real_T A;                            /* Variable: A
                                        */
  real_T B;                            /* Variable: B
                                        */
  real_T SineWave_Amp;                 /* Expression: 1
                                        */
  real_T SineWave_Bias;                /* Expression: 0
                                        */
  real_T SineWave_Freq;                /* Expression: 1
                                        */
  real_T SineWave_Phase;               /* Expression: 0
                                        */
};
```

The generated code algorithm uses that field in the code that represents the block Gain A. In this case, the global structure variable `ex_extModeExample_P` uses the type `P_ex_extModeExample_T_`:

```
ex_extModeExample_B.GainA = ex_extModeExample_P.A * rtb_SineWave;
```

When you change the value of `A` in the base workspace, the simulation downloads the new value to the field `A` in the target program.

You can change the expressions in the Sine Wave block parameters during simulation because the generated code creates a field in the global structure `ex_extModeExample_P` to represent each parameter in the block. When you change an expression in the block dialog box, the simulation first evaluates the new expression. The simulation then downloads the resulting numeric value to the corresponding structure field in the target program.

See “Block Parameter Representation in the Generated Code” (Simulink Coder).

Configure and Use External Mode

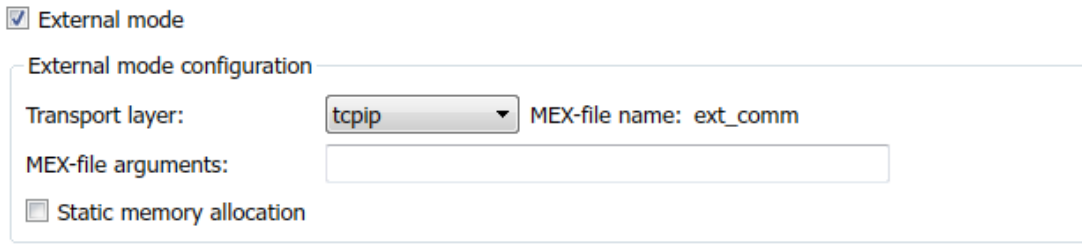
Configure External Mode Options for Code Generation

The list of targets and products that support external mode includes the ERT, GRT, and RSim targets, the Simulink Desktop Real-Time product, and Simulink target hardware platforms. Code generation targets that support external mode provide a set of external mode options in the Configuration Parameters dialog box, on the **Code Generation** > **Interface** pane or their respective target pane.

For targets that support the ability to build target code, connect to the target, and run an application in external mode. Note the following:

- You select external mode from the model window by clicking **Simulation** > **Mode** > **External**.
- External mode parameters continue to appear for the target in the Configuration Parameters dialog box.
- By default, the code generator produces code that is *not* set up for external mode if you build the code by using one of following methods:
  - Press **Ctrl+B**
  - At the MATLAB command line, enter `rtwbuild`

The following figure shows external mode parameters from the GRT and ERT target views of the **Code Generation** > **Interface** pane.



Note The Simulink Real-Time product also uses external mode communication. External mode in the Simulink Real-Time product is always on, and does not have interface options.

The **Data exchange interface** section in the **Code Generation** > **Interface** pane includes the following external mode parameters:

- **External mode** option: Includes the external mode data exchange interface in the generated C code.

  When you select **External mode**, the following parameters appear:

- **Transport layer** menu: Identifies the messaging protocol for host/target communication; typically, the choices are `tcpip` and `serial`.

  The default is `tcpip`. When you select a protocol, the MEX-file name that implements the protocol is shown to the right of the menu.

- **MEX-file arguments** text field: Optionally enter a list of arguments to be passed to the transport layer MEX-file for communicating with executing targets. The arguments vary according to the protocol that you use.

  For more information on the transport options, see “Target Interfacing” on page 41-16 and “Choose Communication Protocol for Client and Server” on page 41-40.

- **Static memory allocation** check box: Controls how memory is allocated for external mode communication buffers in the target. Selecting this option enables the **Static memory buffer size** parameter.

- **Static memory buffer size** text field: Number of bytes to preallocate for external mode communication buffers in the target.

Note Selecting **External mode** does not cause the Simulink model to operate in external mode (see "Control Host and Target Execution" on page 41-18). The **External mode** option instruments the code generated for the target to support external mode.

The **Static memory allocation** check box (for GRT and ERT targets) directs the Simulink Coder software to generate code for external mode that uses only static memory allocation ("malloc-free" code). Selecting **Static memory allocation** enables the **Static memory buffer size** edit field, which you use to specify the size of the static memory buffer used by external mode. The default value is 1,000,000 bytes. If you enter too small a value for your application, external mode issues an out-of-memory error when it tries to allocate more memory than you allowed. In such cases, increase the value in the **Static memory buffer size** field and regenerate the code.

To determine how much memory to allocate, enable verbose mode on the target (by including `OPTS="-DVERBOSE"` on the `make` command line). As it executes, external mode displays the amount of memory it tries to allocate and the amount of memory available to it each time it attempts an allocation. If an allocation fails, you can use this console log to adjust the size in the **Static memory buffer size** field.

Note When you create an ERT target, external mode can generate pure integer code. Select pure integer code by clearing the **Support floating-point numbers** option on the **Code Generation > Interface** pane of the Configuration Parameters dialog box. Clearing this option makes the code, including external mode support code, free of doubles and floats. For more information, see "Model Configuration Parameters: Code Generation Interface" (Simulink Coder).

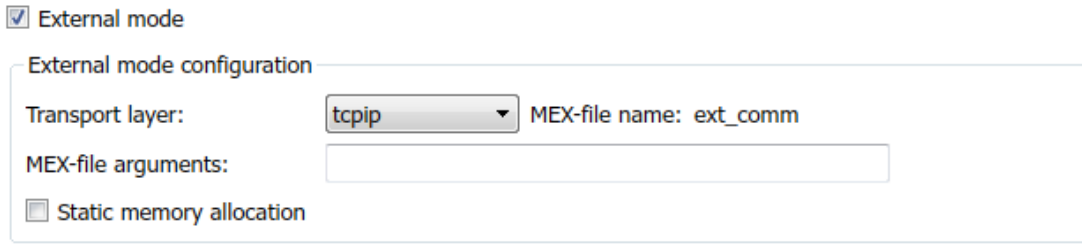
Target Interfacing

The Simulink Coder product lets you implement client and server transport for external mode using either TCP/IP or serial protocols. If your target system supports TCP/IP, you can use the socket-based external mode implementation provided by the Simulink Coder product with the generated code. Otherwise, use or customize the serial transport layer option.

A low-level *transport layer* handles physical transmission of messages. Both the Simulink engine and the model code are independent of this layer. Both the transport layer and

code directly interfacing to the transport layer are isolated in separate modules that format, transmit, and receive messages and data packets.

You specify the transport mechanism using the **Transport layer** menu in the **External mode configuration** section of the **Code Generation** > **Interface** pane of the Configuration Parameters dialog box.



To the right of the **Transport layer** menu, **MEX-file name** displays the name of an external interface MEX-file. This MEX-file implements host/target communication for the selected external mode transport layer. The default is `ext_comm`, the TCP/IP-based external interface file for use with the GRT, ERT, and RSim targets. If you select the `serial` transport option, the MEX-file name `ext_serial_win32_com` is displayed in this location.

Custom or third-party targets can use a custom transport layer and a different external interface MEX-file. For more information, see "Create a Transport Layer for External Communication" (Simulink Coder). For more information on specifying a TCP/IP or serial transport layer for a custom target, see "Using the TCP/IP Implementation" on page 41-41 or "Using the Serial Implementation" on page 41-44.

In the **MEX-file arguments** edit field, you can optionally specify arguments that are passed to the external mode interface MEX-file for communicating with executing targets. The meaning of the MEX-file arguments depends on the MEX-file implementation.

For TCP/IP interfaces, `ext_comm` allows these optional arguments:

- Network name of your target (for example, `'myPuter'` or `'148.27.151.12'`)
- Verbosity level (`0` for no information or `1` for detailed information)
- TCP/IP server port number (an integer value between `256` and `65535`, with a default of `17725`)

For serial transport, `ext_serial_win32_comm` allows these optional arguments:

- Verbosity level (`0` for no information or `1` for detailed information)
- Serial port ID (for example, `1` for `COM1`, and so on)
- Baud rate (selected from the set `1200`, `2400`, `4800`, `9600`, `14400`, `19200`, `38400`, `57600`, and `115200`, with a default baud rate of `57600`)

For more information on MEX-file transport architecture and arguments, see "Choose Communication Protocol for Client and Server" on page 41-40.

Control Host and Target Execution

Simulink software provides multiple ways to control external mode host and target execution. The following table lists common steps in the external mode workflow and the ways in which they can be initiated.

External Mode Action	Toolbar Control	Menu Control	External Mode Control Panel Button
Set the simulation mode of your model to external mode	From the simulation mode drop-down list, select `External`	**Simulation** > **Mode** > **External**	**Connect** (if the model simulation mode is not already set, sets the simulation mode to external mode)
Connect your model to a waiting or running target program	**Connect to Target** button 	**Simulation** > **Connect to Target**	**Connect**
Start running real-time code in the target environment	**Run** button	**Simulation** > **Run** (keyboard shortcut **Ctrl+T**)	**Start Real-Time Code**
Disconnect your model from the target environment (does not halt running real-time code)	**Disconnect from Target** button 	**Simulation** > **Disconnect from Target**	**Disconnect**
Stop target program execution and disconnect your model from the target environment	**Stop** button	**Simulation** > **Stop** (keyboard shortcut **Ctrl+Shift+T**)	**Stop Real-Time Code**

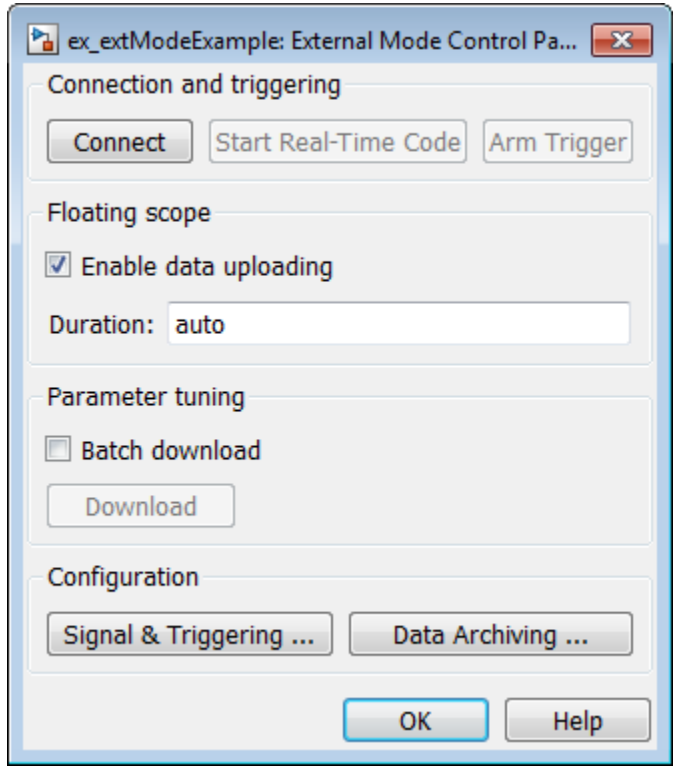
Setting the simulation mode of your model to external mode affects execution only, and does *not* cause the Simulink Coder software to generate code instrumented for external mode. See "Configure External Mode Options for Code Generation" on page 41-14.

Control External Mode Operations

The External Mode Control Panel provides centralized control of external mode operations, including:

- "Connect, Start, and Stop" on page 41-20
- "Upload Target Program Signal Data to Host" on page 41-21
- "Download Parameters to Target Program" on page 41-22
- "Configure Host Monitoring of Target Program Signal Data" on page 41-23
- "Configure Host Archiving of Target Program Signal Data" on page 41-31

To open the External Mode Control Panel dialog box, in the model window, select **Code** > **External Mode Control Panel**.



Connect, Start, and Stop

The External Mode Control Panel performs the same connect/disconnect and start/stop functions found in the **Simulation** menu and the Simulink toolbar (see "Control Host and Target Execution" on page 41-18).

Clicking the **Connect** button connects your model to a waiting or running target program. While you are connected, the button changes to a **Disconnect** button. **Disconnect** disconnects your model from the target environment, but does not halt real-time code running in the target environment.

Connect sets the model simulation mode to external mode.

Clicking the **Start Real-Time Code** button commands the target to start running real-time code. While real-time code is running in the target environment, the button changes

to a **Stop Real-Time Code** button. **Stop Real-Time Code** stops target program execution and disconnects your model from the target environment.

Upload Target Program Signal Data to Host

The External Mode Control Panel allows you to trigger and cancel data uploads to the host. The destination for the uploaded data can be a scope block, Display block, To Workspace block, or another block or subsystem listed in "External Mode Compatible Blocks and Subsystems" on page 41-34.

The **Arm Trigger** and **Cancel Trigger** buttons provide manual control of data uploading to compatible blocks or subsystems, except floating scopes. (For floating scopes, use the **Floating scope** section of the External Mode Control Panel.)

- To trigger data uploading to compatible blocks or subsystems, click the **Arm Trigger** button. The button changes to **Cancel Trigger**.
- To cancel data uploading, click the **Cancel Trigger** button. The button reverts to **Arm Trigger**.

You can trigger data uploads manually or automatically. To configure signals and triggers for data uploads, see "Configure Host Monitoring of Target Program Signal Data" on page 41-23.

A subset of external mode compatible blocks, including Scope, Time Scope, and To Workspace, allow you to log uploaded data to disk. To configure data archiving, see "Configure Host Archiving of Target Program Signal Data" on page 41-31.

The **Floating scope** section of the External Mode Control Panel controls when and for how long data is uploaded to Floating Scope blocks. When used in external mode, floating scopes:

- Do not appear in the External Signal & Triggering dialog box.
- Do not log data to external mode archiving.
- Support manual triggering only.

The **Floating scope** section contains the following parameters:

- **Enable data uploading** option, which functions as an **Arm Trigger** button for floating scopes. When the target is disconnected, the option controls whether to arm the trigger when connecting the floating scopes. When the target is connected, the option acts as a toggle button to arm or cancel the trigger.

  - To trigger data uploading to floating scopes, select **Enable data uploading**.
  - To cancel data uploading to floating scopes, clear **Enable data uploading**.
- **Duration** edit field, which specifies the number of base-rate steps for which external mode logs floating scopes data after a trigger event. By default, it is set to `auto`, which causes the duration value set in the External Signal & Triggering dialog box to be used. The default duration value is 1000 base rate steps.

Download Parameters to Target Program

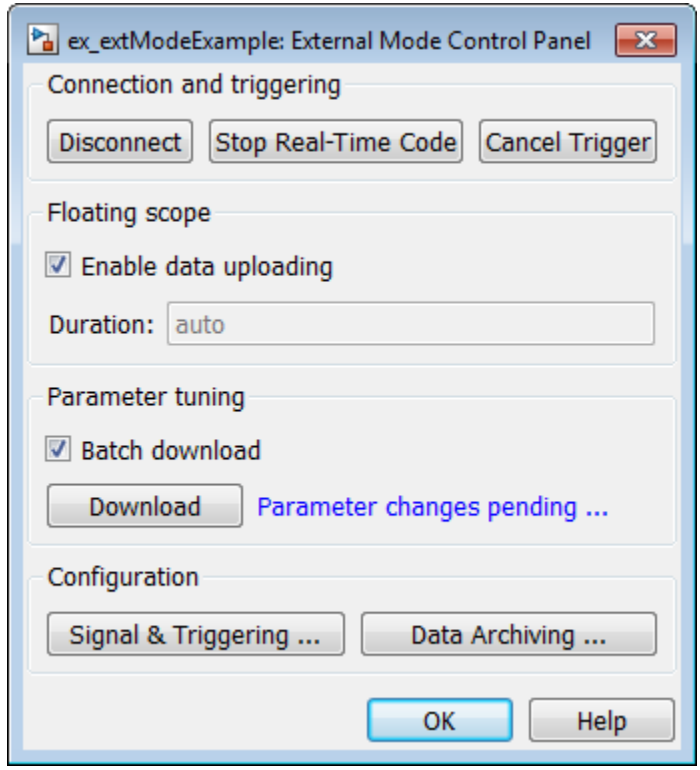
The **Batch download** option on the External Mode Control Panel enables or disables batch parameter changes.

By default, batch download is disabled. If batch download is disabled, when you click **OK** or **Apply**, changes made directly to block parameters by editing block parameter dialog boxes are sent to the target. When you perform an **Update Diagram**, changes to MATLAB workspace variables are sent.

If you select **Batch download**, the **Download** button is enabled. Until you click **Download**, changes made to block parameters are stored locally. When you click **Download**, the changes are sent in a single transmission.

When parameter changes are awaiting batch download, the External Mode Control Panel displays the message `Parameter changes pending...` to the right of the **Download** button. This message remains visible until the Simulink engine receives notification that the new parameters have been installed in the parameter vector of the target system.

The next figure shows the External Mode Control Panel with the **Batch download** option activated and parameter changes pending.



Configure Host Monitoring of Target Program Signal Data

Role of Trigger in Signal Data Uploading

In external mode, uploading target program signal data to the host depends on a *trigger*. The trigger is a set of conditions that must be met for data uploading to begin. The trigger can be armed or not armed.

- When the trigger is armed, the software checks for the trigger conditions that allow data uploading to begin.
- If the trigger is not armed, the software does not check for the trigger conditions and data uploading cannot begin.
- The trigger can be armed automatically, when the host connects to the target, or manually, by clicking the **Arm Trigger** button on the External Mode Control Panel.

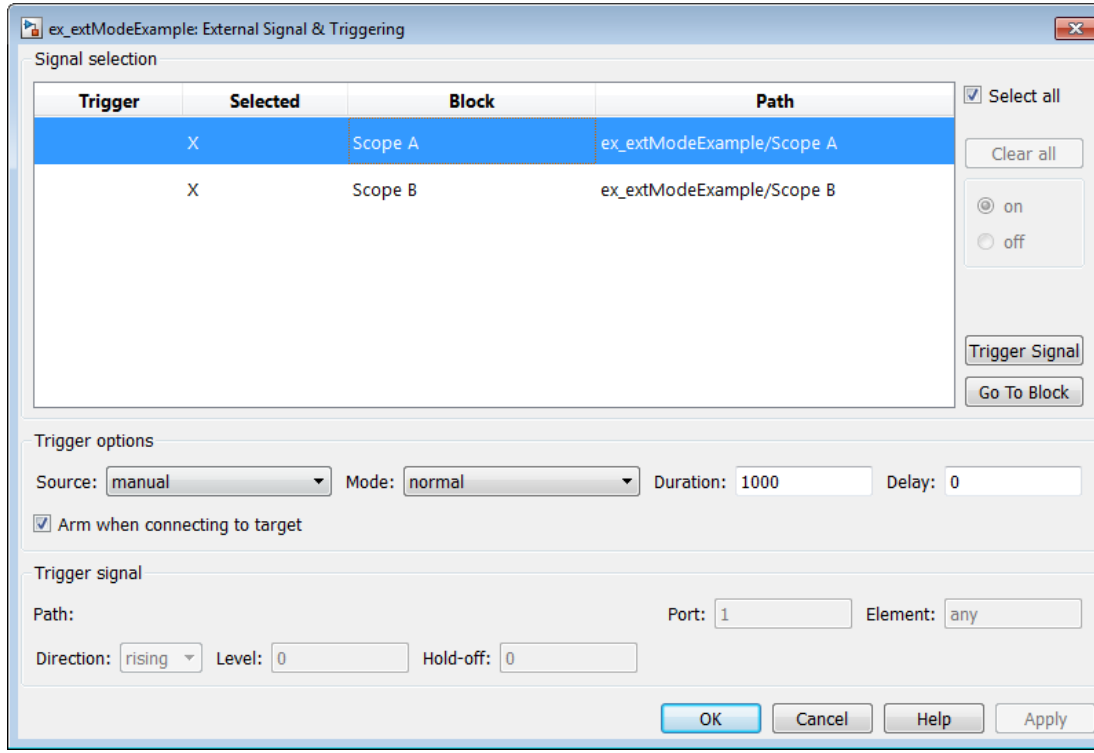
When the trigger is armed and the trigger conditions are met, the trigger fires and data uploading begins.

When data has been collected for a defined duration, the trigger event completes and data uploading stops. The trigger can then rearm, or remain unarmed until you click the **Arm Trigger** button.

To select the target program signals to upload and configure how uploads are triggered, see “Configure Signal Data Uploading” on page 41-24.

Configure Signal Data Uploading

Clicking the **Signal & Triggering** button of the External Mode Control Panel opens the External Signal & Triggering dialog box.



The External Signal & Triggering dialog box displays a list of blocks and subsystems in your model that support external mode signal uploading. For information on which types of blocks are external mode compatible, see “External Mode Compatible Blocks and Subsystems” on page 41-34.

In the External Signal & Triggering dialog box, you can select the signals that are collected from the target system and viewed in external mode. You can also select a trigger signal, which triggers uploading of data based on meeting certain signal conditions, and define the triggering conditions.

Default Trigger Options

The preceding figure shows the default settings of the External Signal & Triggering dialog box. The default operation of the External Signal & Triggering dialog box simplifies monitoring the target program. If you use the default settings, you do not need to preconfigure signals and triggers. You start the target program and connect the

Simulink engine to it. External mode compatible blocks are selected and the trigger is armed. Signal uploading begins immediately upon connection to the target program.

The default configuration of trigger options is:

- `Select all`: on
- `Source`: manual
- `Mode`: normal
- `Duration`: 1000
- `Delay`: 0
- `Arm when connecting to target`: on

Select Signals to Upload

External mode compatible blocks in your model appear in the **Signal selection** list of the External Signal & Triggering dialog box. You use this list to select signals that you want to view. In the **Selected** column, an `X` appears for each selected block.

The **Select all** check box selects all signals. By default, **Select all** is selected.

If **Select all** is cleared, you can select or clear individual signals using the **on** and **off** options. To select a signal, click its list entry and select the **on** option. To clear a signal, click its list entry and select the **off** option.

The **Clear all** button clears all signals.

Configure Trigger Options

As described in "Role of Trigger in Signal Data Uploading" on page 41-24, signal data uploading depends on a trigger. The trigger defines conditions that must be met for uploading to begin. Also, the trigger must be armed for data uploading to begin. When the trigger is armed and trigger conditions are met, the trigger fires and uploading begins. When data has been collected for a defined duration, the trigger event completes and data uploading stops.

To control when and how signal data is collected (uploaded) from the target system, configure the following **Trigger options** in the External Signal & Triggering dialog box.

- **Source**: `manual` or `signal`. Controls whether a button or a signal triggers data uploading.

Selecting `manual` directs external mode to use the **Arm Trigger** button on the External Mode Control Panel as the trigger to start uploading data. When you click **Arm Trigger**, data uploading begins.

Selecting `signal` directs external mode to use a trigger signal as the trigger to start uploading data. When the trigger signal satisfies trigger conditions (that is, the signal crosses the trigger level in the specified direction), a *trigger event* occurs. (Specify trigger conditions in the **Trigger signal** section.) If the trigger is *armed*, external mode monitors for the occurrence of a trigger event. When a trigger event occurs, data uploading begins.

- **Mode**: `normal` or `one-shot`. Controls whether the trigger rearms after a trigger event completes.

  In `normal` mode, external mode automatically rearms the trigger after each trigger event. The next data upload begins when the trigger fires.

  In `one-shot` mode, external mode collects only one buffer of data each time you arm the trigger.

  For more information on the **Mode** setting, see "Configure Host Archiving of Target Program Signal Data" on page 41-31.
- **Duration**: Specifies the number of base rate steps for which external mode uploads data after a trigger event (default is 1000). For example, if **Duration** is set to 1000, and the base (fastest) rate of the model is one second:
  - For a signal sampled at the base rate, one second (1.0 Hz), external mode collects 1000 contiguous samples during a trigger event.
  - For a signal sampled at two seconds (0.5 Hz), external mode collects 500 samples during a trigger event.
- **Delay**: Specifies a delay to be applied to data collection. The delay represents the amount of time that elapses between a trigger event and the start of data collection. The delay is expressed in base rate steps. It can be positive or negative (default is 0). A negative delay corresponds to pretriggering. When the delay is negative, data from the time preceding the trigger event is collected and uploaded.
- **Arm when connecting to target**: Selected or cleared. Whether a button or a signal triggers data uploading (as defined by **Source**), the trigger must be armed to allow data uploading to begin.

  If you select this option, connecting to the target arms the trigger.

- If the trigger **Source** is `manual`, data uploading begins immediately.
- If the trigger **Source** is `signal`, monitoring of the trigger signal begins immediately. Data uploading begins when the trigger signal satisfies trigger conditions (as defined in the **Trigger signal** section).

If you clear **Arm when connecting to target**, manually arm the trigger by clicking the **Arm Trigger** button on the External Mode Control Panel.

When simulating in external mode, each rate in the model creates a buffer on the target. Each entry in the buffer is big enough to hold all of the data required of every signal in that rate for one time step (time plus data plus external mode indices identifying the signal). The number of entries in the circular buffer is determined by the external mode trigger **Duration** parameter (`ExtModeTrigDuration`). The memory allocated on the target for buffering signals is proportional to the **Duration** and the number of signals uploading. The **Duration** also provides an indication of the number of base rate steps with log data after a trigger event in external mode.

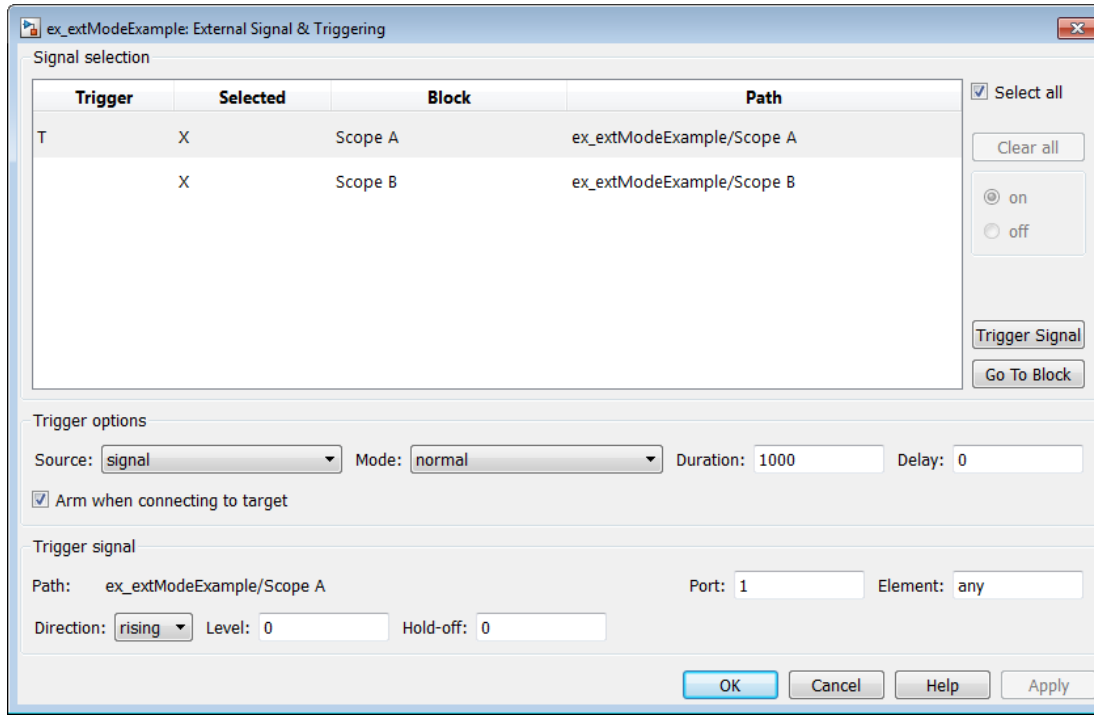
The **Duration** value specifies the number of contiguous points of data to be collected in each buffer of data. You should enter a **Duration** value equal to the number of continuous sample points that you need to collect rather than relying on a series of buffers to be continuous. If you enter a value less than the total number of sample points, you may lose sample points during the time spent transferring values from the data buffer to the MATLAB workspace. The Simulink software maintains point continuity only within one buffer. Between buffers, because of transfer time, some samples may be omitted.

The **Duration** value can affect the **Limit data points to last** value of Scope and To Workspace blocks. The number of sample points that the blocks save to the MATLAB workspace is the smaller of the two values. To set the number of sample points that the blocks save, clear **Limit data points to last**. Then, use **Duration** to specify the number of sample points saved.

Select Trigger Signal

You can designate one signal as a trigger signal. To select a trigger signal, from the **Source** menu in the **Trigger options** section, select `signal`. This action enables the parameters in the **Trigger signal** section. Then, select a signal in the **Signal selection** list, and click the **Trigger Signal** button.

When you select a signal to be a trigger, a `T` appears in the **Trigger** column of the **Signal selection** list. In the next figure, the `Scope A` signal is the trigger. `Scope B` is also selected for viewing, as indicated by the `X` in the **Selected** column.



After selecting the trigger signal, you can use the **Trigger signal** section to define the trigger conditions and set the trigger signal **Port** and **Element** parameters.

Set Trigger Conditions

Use the **Trigger signal** section of the External Signal & Triggering dialog box to set trigger conditions and attributes. **Trigger signal** parameters are enabled only when the trigger parameter **Source** is set to `signal` in the **Trigger options** section.

By default, any element of the first input port of a specified trigger block can cause the trigger to fire (that is, Port 1, any element). You can modify this behavior by adjusting the **Port** and **Element** values in the **Trigger signal** section. The **Port** field accepts a number or the keyword `last`. The **Element** field accepts a number or the keywords `any` or `last`.

In the **Trigger signal** section, you also define the conditions under which a trigger event occurs.

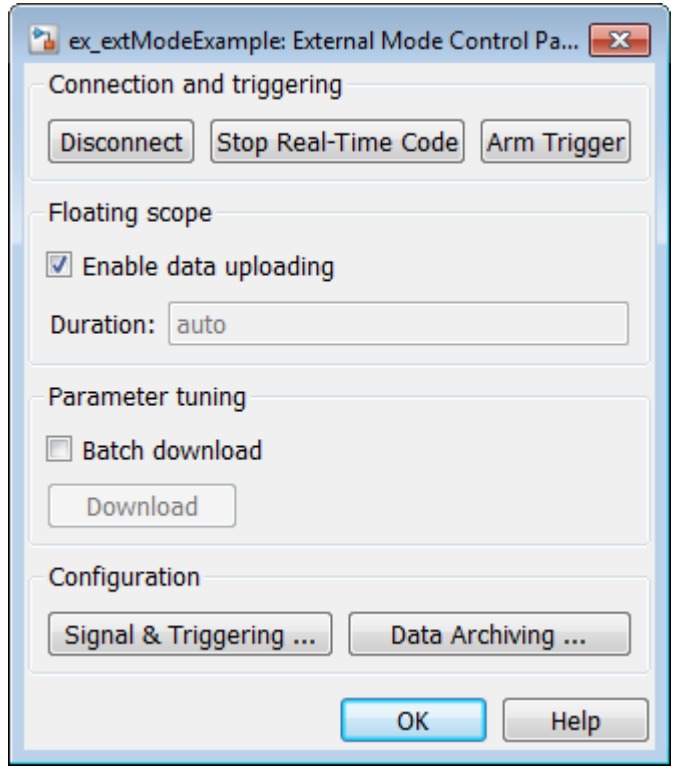
- **Direction**: `rising`, `falling`, or `either`. The direction in which the signal must be traveling when it crosses the threshold value. The default is `rising`.
- **Level**: A value indicating the threshold the signal must cross in a designated direction to fire the trigger. By default, the level is 0.
- **Hold-off**: Applies only to `normal` mode. Expressed in base rate steps, **Hold-off** is the time between the termination of one trigger event and the rearming of the trigger.

Modify Signal and Triggering Options While Connected

After you configure signal data uploading, and connect Simulink to a running target executable, you can modify signal and triggering options without disconnecting from the target.

If the trigger is armed (for example, if the trigger option **Arm when connecting to the target** is selected, which is the default), the External Signal & Triggering dialog box cannot be modified. To modify signal and triggering options:

1. Open the External Mode Control Panel.
2. Click **Cancel Trigger**. Triggering and display of uploaded data stops.
3. Open the External Signal & Triggering dialog box and modify signal and trigger options as required. For example, in the **Signal selection** section, you can enable or disable a scope, and in the **Trigger options** section, change the trigger **Mode**, for example, from `normal` to `one-shot`.
4. Click **Arm Trigger**. Triggering and display of uploaded data resumes, with your modifications.



Configure Host Archiving of Target Program Signal Data

In external mode, you can use the Simulink Scope and To Workspace blocks to archive data to disk.

To understand how the archiving features work, consider the handling of data when archiving is not enabled. There are two cases, one-shot mode and normal mode.

- In one-shot mode, after a trigger event occurs, each selected block writes its data to the workspace, as it would at the end of a simulation. If another one-shot is triggered, the existing workspace data is overwritten.
- In normal mode, external mode automatically rearms the trigger after each trigger event. Consequently, you can think of normal mode as a series of one-shots. Each one-shot in this series, except for the last, is referred to as an *intermediate result*. Because

the trigger can fire at any time, writing intermediate results to the workspace can result in unpredictable overwriting of the workspace variables. For this reason, the default behavior is to write only the results from the final one-shot to the workspace. The intermediate results are discarded. If you know that enough time exists between triggers for inspection of the intermediate results, you can override the default behavior by selecting the **Write intermediate results to workspace** option. This option does not protect the workspace data from being overwritten by subsequent triggers.

If you use a Simulink Scope block to archive data to disk, open the Scope parameters dialog box and select the option **Log data to workspace**. The option is required for these reasons:

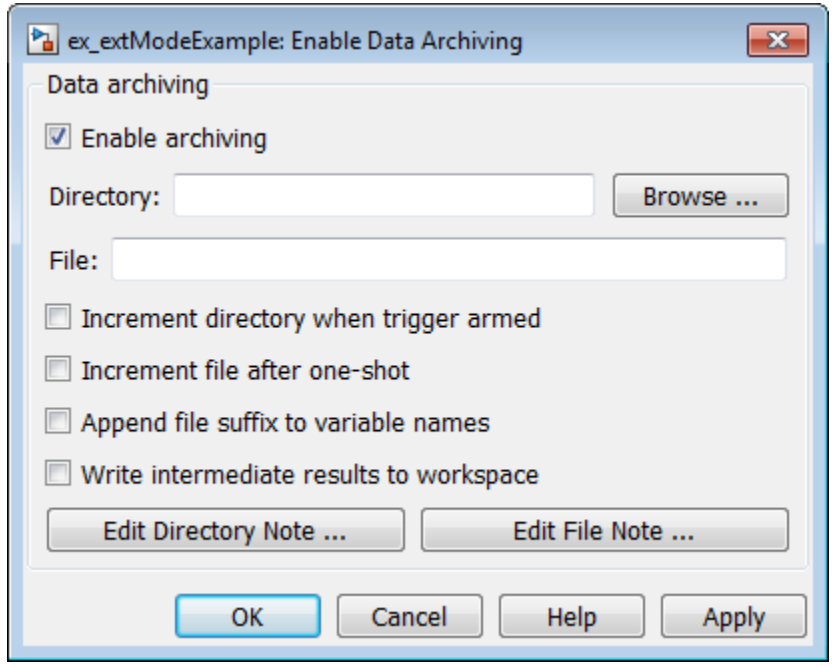
- The data is first transferred from the scope data buffer to the MATLAB workspace, before being written to a MAT-file.
- The **Variable name** entered in the Scope parameters dialog box is the same as the one in the MATLAB workspace and the MAT-file. Enabling the data to be saved enables a variable named with the **Variable name** parameter to be saved to a MAT-file.

Note: If you do not select the Scope block option **Log data to workspace**, the MAT-files for data logging are created, but they are empty.

The Enable Data Archiving dialog box supports:

- Folder notes
- File notes
- Automated data archiving

On the External Mode Control Panel, click the **Data Archiving** button to open the Enable Data Archiving dialog box. If your model is connected to the target environment, disconnect it while you configure data archiving. To enable the other controls in the dialog box, select **Enable archiving**.



These operations are supported by the Enable Data Archiving dialog box.

Folder Notes

To add annotations for a collection of related data files in a folder, in the Enable Data Archiving dialog box, click **Edit Directory Note**. The MATLAB editor opens. Place comments that you want saved to a file in the specified folder in this window. By default, the comments are saved to the folder last written to by data archiving.

File Notes

To add annotations for an individual data file, in the Enable Data Archiving dialog box, click **Edit File Note**. A file finder window opens, which by default is set to the last file to which you have written. Selecting a MAT-file opens an edit window. In this window, add or edit comments that you want saved with your individual MAT-file.

Automated Data Archiving

To configure automatic writing of logging results to disk, optionally including intermediate results, use the **Enable archiving** option and the controls it enables . The dialog box provides the following related controls:

- **Directory**: Specifies the folder in which data is saved. If you select **Increment directory when trigger armed**, external mode appends a suffix.
- **File**: Specifies the name of the file in which data is saved. If you select **Increment file after one-shot**, external mode appends a suffix.
- **Increment directory when trigger armed**: Each time that you click the **Arm Trigger** button, external mode uses a different folder for writing log files. The folders are named incrementally, for example, `dirname1`, `dirname2`, and so on.
- **Increment file after one-shot**: New data buffers are saved in incremental files: `filename1`, `filename2`, and so on. File incrementing happens automatically in normal mode.
- **Append file suffix to variable names**: Whenever external mode increments file names, each file contains variables with identical names. Selecting **Append file suffix to variable name** results in each file containing unique variable names. For example, external mode saves a variable named `xdata` in incremental files (`file_1`, `file_2`, and so on) as `xdata_1`, `xdata_2`, and so on. This approach supports loading the MAT-files into the workspace and comparing variables at the MATLAB command prompt. Without the unique names, each instance of `xdata` would overwrite the previous one in the MATLAB workspace.
- **Write intermediate results to workspace**: If you want the Simulink Coder software to write intermediate results to the workspace, select this option.

External Mode Compatible Blocks and Subsystems

Compatible Blocks

In external mode, you can use the following types of blocks to receive and view signals uploaded from the target program:

- Floating Scope and Scope blocks
- Spectrum Analyzer, Time Scope, and Vector Scope blocks from the DSP System Toolbox product
- Display blocks

- To Workspace blocks
- User-written S-Function blocks

  An external mode method is built into the S-function API. This method allows user-written blocks to support external mode. See *`matlabroot`*`/simulink/include/simstruc.h`.
- XY Graph blocks

You can designate certain subsystems as Signal Viewing Subsystems and use them to receive and view signals uploaded from the target program. See "Signal Viewing Subsystems" on page 41-35 for more information.

You select external mode compatible blocks and subsystems, and arm the trigger, by using the External Signal & Triggering dialog box. By default, such blocks in a model are selected, and a manual trigger is set to be armed when connected to the target program.

Signal Viewing Subsystems

A Signal Viewing Subsystem is an atomic subsystem that encapsulates processing and viewing of signals received from the target system. A Signal Viewing Subsystem runs only on the host, and does not generate code in the target system. Signal Viewing Subsystems run in normal, accelerator, rapid accelerator, and external simulation modes.

Note: Signal Viewing Subsystems are inactive if placed inside a SIL or PIL component, such as a top model in SIL or PIL mode, a Model block in SIL or PIL mode, or a SIL or PIL block. However, a SIL or PIL component can feed a Signal Viewing Subsystem running in a supported mode.

Signal Viewing Subsystems are useful in situations where you want to process or condition signals before viewing or logging them, but you do not want to perform these tasks on the target system. By using a Signal Viewing Subsystem, you can generate smaller and more efficient code on the target system.

Like other external mode compatible blocks, Signal Viewing Subsystems are displayed in the External Signal & Triggering dialog box.

To declare a subsystem to be a Signal Viewing Subsystem:

1 In the Block Parameters dialog box, select the **Treat as atomic unit** option.

For more information on atomic subsystems, see "Code Generation of Subsystems" (Simulink Coder).

2 To turn the SimViewingDevice property on, use the `set_param` command:

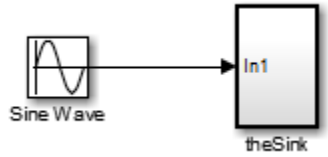
```
set_param('blockname', 'SimViewingDevice','on')
```

`'blockname'` is the name of the subsystem.

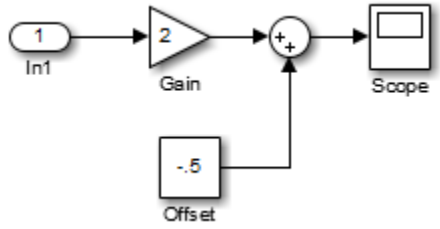
3 Make sure the subsystem meets the following requirements:

- It must be a pure Sink block. That is, it must not contain Outport blocks or Data Store blocks. It can contain Goto blocks only if the corresponding From blocks are contained within the subsystem boundaries.
- It must not have continuous states.

The following model, `sink_examp`, contains an atomic subsystem, `theSink`.



The subsystem `theSink` applies a gain and an offset to its input signal and displays it on a Scope block.



If `theSink` is declared as a Signal Viewing Subsystem, the generated target program includes only the code for the Sine Wave block. If `theSink` is selected and armed in the External Signal & Triggering dialog box, the target program uploads the sine wave

signal to `theSink` during simulation. You can then modify the parameters of the blocks within `theSink` and observe the uploaded signal.

If `theSink` were not declared as a Signal Viewing Subsystem, its Gain, Constant, and Sum blocks would run as subsystem code on the target system. The Sine Wave signal would be uploaded to the Simulink engine after being processed by these blocks, and viewed on `sink_examp/theSink/Scope2`. Processing demands on the target system would be increased by the additional signal processing, and by the downloading of changes in block parameters from the host.

Supported Blocks for Data Archiving

In external mode, you can use the following types of blocks to archive data to disk:

- Scope blocks
- To Workspace blocks

You configure data archiving by using the Enable Data Archiving dialog box, as described in "Configure Host Archiving of Target Program Signal Data" on page 41-31.

External Mode Communication

About External Mode Communication

Depending on the setting of the **Default parameter behavior** option when the target program is generated, there are differences in the way parameter updates are handled. "Download Mechanism" on page 41-37 describes the operation of external mode communication with **Default parameter behavior** set to `Tunable`. "Inlined and Tunable Parameters" on page 41-39 describes the operation of external mode with **Default parameter behavior** set to `Inlined`.

Download Mechanism

In external mode, the Simulink engine does not simulate the system represented by the block diagram. By default, when external mode is enabled, the Simulink engine

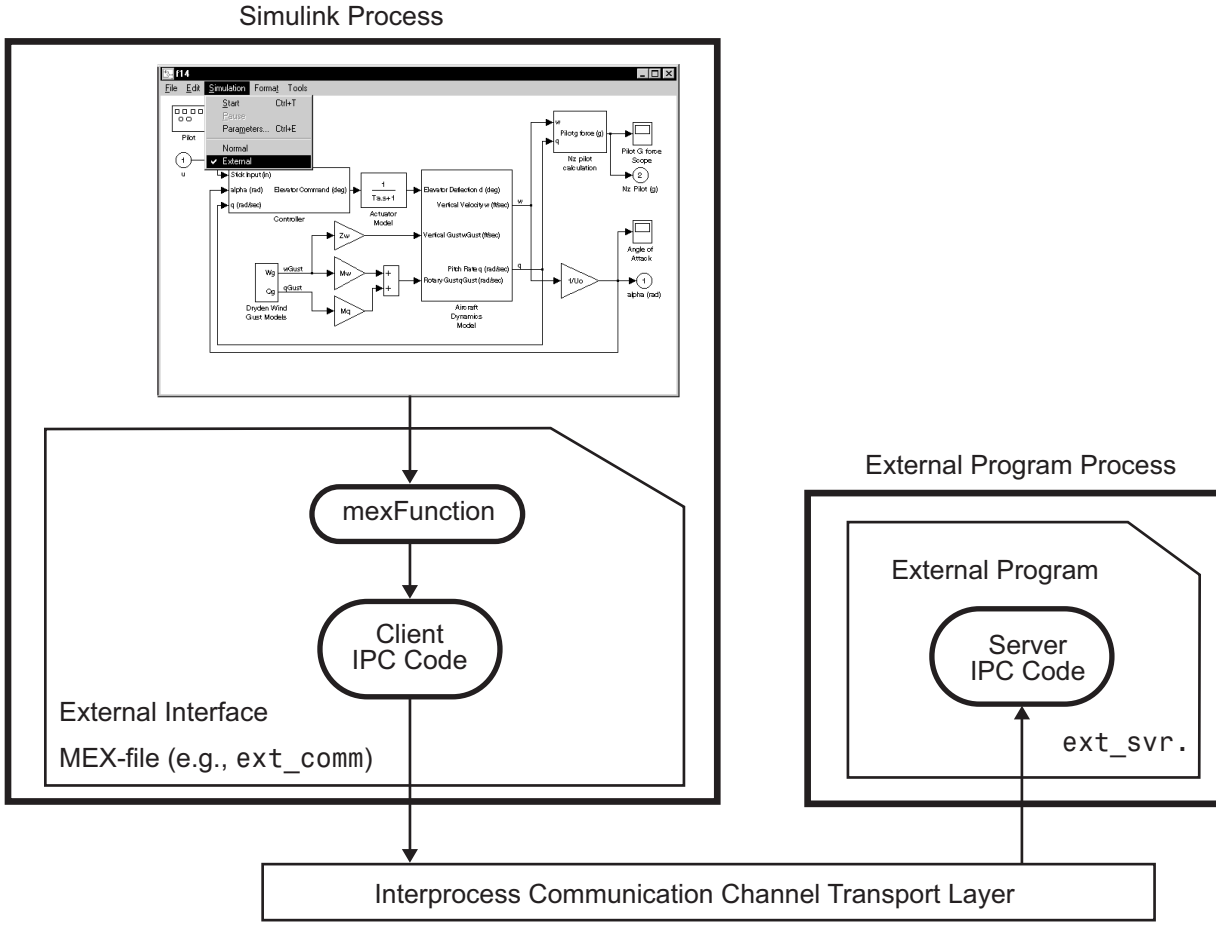
downloads parameters to the target system. After the initial download, the engine remains in a waiting mode until you change parameters in the block diagram or until the engine receives data from the target.

When you change a parameter in the block diagram, the Simulink engine calls the external interface MEX-file, passing new parameter values (along with other information) as arguments. The external interface MEX-file contains code that implements one side of the interprocess communication (IPC) channel. This channel connects the Simulink process (where the MEX-file executes) to the process that is executing the external program. The MEX-file transfers the new parameter values by using this channel to the external program.

The other side of the communication channel is implemented within the external program. This side writes the new parameter values into the target's parameter structure (`model_P`).

The Simulink side initiates the parameter download operation by sending a message containing parameter information to the external program. In the terminology of client/server computing, the Simulink side is the client and the external program is the server. The two processes can be remote, or they can be local. Where the client and server are remote, a protocol such as TCP/IP is used to transfer data. Where the client and server are local, a serial connection or shared memory can be used to transfer data.

The next figure shows this relationship. The Simulink engine calls the external interface MEX-file whenever you change parameters in the block diagram. The MEX-file then downloads the parameters to the external program by using the communication channel.



External Mode Architecture

Inlined and Tunable Parameters

By default, parameters (except those listed in "External Mode Limitations" on page 41-55) in an external mode program are tunable; that is, you can change them by using the download mechanism described in this section.

If you set **Default parameter behavior** to `Inlined` (on the **Optimization > Signals and Parameters** pane of the Configuration Parameters dialog box), the Simulink Coder code generator embeds the numerical values of model parameters (constants), instead of

symbolic parameter names, in the generated code. Inlining parameters generates smaller and more efficient code. However, inlined parameters, because they effectively become constants, are not tunable.

The Simulink Coder software lets you improve overall efficiency by inlining most parameters, while at the same time retaining the flexibility of run-time tuning for selected parameters that are important to your application. When you inline parameters, you can use `Simulink.Parameter` objects to remove individual parameters from inlining and declare them to be tunable. In addition, you can use these objects to control how parameters are represented in the generated code.

For more information on tunable parameters, see "Block Parameter Representation in the Generated Code" (Simulink Coder).

Automatic Parameter Uploading on Host/Target Connection

Each time the Simulink engine connects to a target program that was generated with **Default parameter behavior** set to `Inlined`, the target program uploads the current value of its tunable parameters to the host. These values are assigned to the corresponding MATLAB workspace variables. This procedure synchronizes the host and target with respect to parameter values.

Workspace variables required by the model must be initialized at the time of host/target connection. Otherwise the uploading cannot proceed and an error results. Once the connection is made, these variables are updated to reflect the current parameter values on the target system.

Automatic parameter uploading takes place only if the target program was generated with **Default parameter behavior** set to `Inlined`. "Download Mechanism" on page 41-37 describes the operation of external mode communication with **Default parameter behavior** set to `Tunable`.

Choose Communication Protocol for Client and Server

Introduction

The Simulink Coder product provides code to implement both the client and server side of external mode communication using either TCP/IP or serial protocols. You can use the socket-based external mode implementation provided by the Simulink Coder product with the generated code, provided that your target system supports TCP/IP. If not, use or customize the serial transport layer option provided.

A low-level *transport layer* handles physical transmission of messages. Both the Simulink engine and the model code are independent of this layer. Both the transport layer and code directly interfacing to the transport layer are isolated in separate modules that format, transmit, and receive messages and data packets.

For information on selecting a transport layer, see "Target Interfacing" on page 41-16.

Using the TCP/IP Implementation

You can use TCP/IP-based client/server implementation of external mode with real-time programs on The Open Group UNIX or PC systems. For help in customizing external mode transport layers, see "Create a Transport Layer for External Communication" (Simulink Coder).

To use Simulink external mode over TCP/IP:

- Make sure that the external interface MEX-file for your target's TCP/IP transport is specified.

  Targets provided by MathWorks specify the name of the external interface MEX-file in `matlabroot/toolbox/simulink/simulink/extmode_transports.m`. The name of the interface appears as uneditable text in the **External mode configuration** section of the **Interface** pane of the Configuration Parameters dialog box. The TCP/IP default is `ext_comm`.

  To specify a TCP/IP transport for a custom target, you must add an entry of the following form to an `sl_customization.m` file on the MATLAB path:

```
function sl_customization(cm)
  cm.ExtModeTransports.add('stf.tlc', 'transport', 'mexfile', 'Level1');
%end function
```

  - *`stf`*`.tlc` is the name of the system target file for which you are registering the transport (for example, `'mytarget.tlc'`)

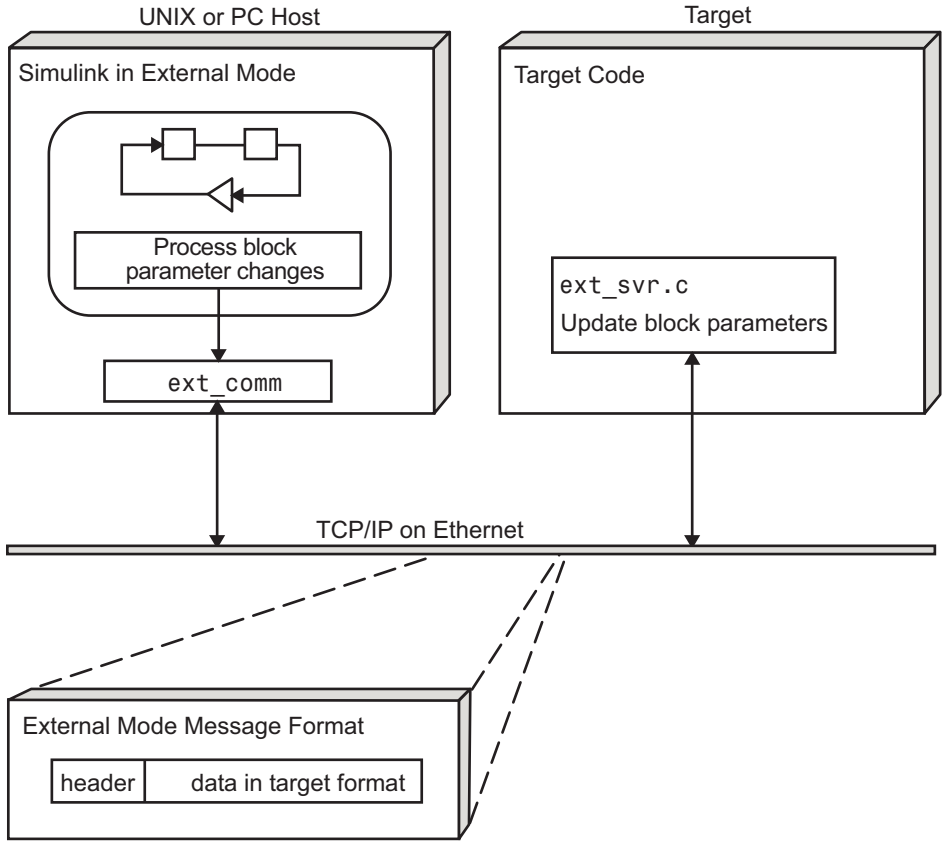
- *`transport`* is the transport name to display in the **Transport layer** menu on the **Interface** pane of the Configuration Parameters dialog box (for example, `'tcpip'`)
- *`mexfile`* is the name of the transport's associated external interface MEX-file (for example, `'ext_comm'`)

You can specify multiple targets and/or transports with additional `cm.ExtModeTransports.add` lines, for example:

```
function sl_customization(cm)
  cm.ExtModeTransports.add('mytarget.tlc', 'tcpip', 'ext_comm', 'Level1');
  cm.ExtModeTransports.add('mytarget.tlc', 'serial', ...
                           'ext_serial_win32_comm', 'Level1');
%end function
```

- Be sure that the template makefile is configured to link the source files for the TCP/IP server code and that it defines the compiler flags when building the generated code.
- Build the external program.
- Run the external program.
- Set the Simulink model to external mode and connect to the target.

The next figure shows the structure of the TCP/IP-based implementation.



TCP/IP-Based Client/Server Implementation for External Mode

MEX-File Optional Arguments for TCP/IP Transport

In the External Target Interface dialog box, you can specify optional arguments that are passed to the external mode interface MEX-file for communicating with executing targets.

- Target network name: the network name of the computer running the external program. By default, this is the computer on which the Simulink product is running, for example, `'myComputer'`. You can also use the IP address, for example, `'148.27.151.12'`.

- Verbosity level: controls the level of detail of the information displayed during the data transfer. The value is either `0` or `1` and has the following meaning:

  `0` — No information
  `1` — Detailed information
- TCP/IP server port number: The default value is `17725`. You can change the port number to a value between `256` and `65535` to avoid a port conflict.

The arguments are positional and must be specified in the following order:

```
<TargetNetworkName> <VerbosityLevel> <ServerPortNumber>
```

For example, if you want to specify the verbosity level (the second argument), then you must also specify the target network name (the first argument). Arguments can be delimited by white space or commas. For example:

```
'148.27.151.12' 1 30000
```

You can specify command-line options to the external program when you launch it. See "Run the External Program" on page 41-46.

Using the Serial Implementation

Controlling host/target communication on a serial channel is similar to controlling host/target communication on a TCP/IP channel.

To use Simulink external mode over a serial channel, you must:

- Make sure that the external interface MEX-file for your target's serial transport is specified.

  Targets provided by MathWorks specify the name of the external interface MEX-file in *`matlabroot`*`/toolbox/simulink/simulink/extmode_transports.m`. The name of the interface appears as uneditable text in the **External mode configuration** section of the **Interface** pane of the Configuration Parameters dialog box. The serial default is `serial`.

  To specify a serial transport for a custom target, you must add an entry of the following form to an `sl_customization.m` file on the MATLAB path:

  ```
  function sl_customization(cm)
    cm.ExtModeTransports.add('stf.tlc', 'transport', 'mexfile', 'Level1');
  %end function
  ```

- *`stf`*`.tlc` is the name of the system target file for which you are registering the transport (for example, `'mytarget.tlc'`)
- *`transport`* is the transport name to display in the **Transport layer** menu on the **Interface** pane of the Configuration Parameters dialog box (for example, `'serial'`)
- *`mexfile`* is the name of the transport's associated external interface MEX-file (for example, `'ext_serial_win32_comm'`)

You can specify multiple targets and/or transports with additional `cm.ExtModeTransports.add` lines, for example:

```
function sl_customization(cm)
  cm.ExtModeTransports.add('mytarget.tlc', 'tcpip', 'ext_comm', 'Level1');
  cm.ExtModeTransports.add('mytarget.tlc', 'serial', ...
                           'ext_serial_win32_comm', 'Level1');
%end function
```

- Be sure that the template makefile is configured to link the source files for the serial server code and that it defines the compiler flags when building the generated code.
- Build the external program.
- Run the external program.
- Set the Simulink model to external mode and connect to the target.

MEX-File Optional Arguments for Serial Transport

In the **MEX-file arguments** field of the **Interface** pane of the Configuration Parameters dialog box, you can specify arguments that are passed to the external mode interface MEX-file for communicating with the executing targets. For serial transport, the optional arguments to `ext_serial_win32_comm` are as follows:

- Verbosity level: This argument controls the level of detail of the information displayed during data transfer. The value of this argument is:
  - `0` (no information), or
  - `1` (detailed information)
- Serial port ID: The port ID of the host, specified as an integer or character vector. For example, specify the port ID of a USB to serial converter as `'/dev/ttyusb0'`.Simulink Coder prefixes integer port IDs with `\\.\COM` on Windows and by `/dev/ttyS` on Unix.

When you start the target program using a serial connection, you must specify the port ID to use to connect it to the host. Do this by including the `-port` command-line option. For example:

```
mytarget.exe -port 2 -w
```

- Baud rate: Specify an integer value. The default value is `57600`.

The MEX-file options arguments are positional and must be specified in the following order:

```
<VerbosityLevel> <SerialPortID> <BaudRate>
```

For example, if you want to specify the serial port ID (the second argument), then you must also specify the verbosity level (the first argument). Arguments can be delimited by white space or commas. For example:

```
1 '/dev/ttyusb0' 57600
```

When you launch the external program, you can specify command-line options.

Run the External Program

Before you can use the Simulink product in external mode, the external program must be running.

If the target program is executing on the same machine as the host and communication is through a loopback serial cable, the target's port ID must differ from that of the host (as specified in the **MEX-file arguments** edit field).

To run the external program, you type a command of the form:

```
model -opt1 ... -optN
```

`model` is the name of the external program and *`-opt1 ... -optN`* are options. (See "Command-Line Options for the External Program" on page 41-47.) In the examples in this section, the name of the external program is `ext_example`.

Running the External Program in the Windows Environment

In the Windows environment, you can run the external programs in either of the following ways:

- Open a Command Prompt window. At the command prompt, type the name of the target executable, followed by possible options, such as:

```
ext_example -tf inf -w
```

- Alternatively, you can launch the target executable from the MATLAB Command Window. The command must be preceded by an exclamation point (`!`) and followed by an ampersand (`&`), as in the following example:

  ```
  !ext_example -tf inf -w &
  ```

  The ampersand (`&`) causes the operating system to spawn another process to run the target executable. If you do not include the ampersand, the program still runs, but you cannot enter commands at the MATLAB command prompt or manually terminate the executable.

Running the External Program in the UNIX Environment

In the UNIX environment, you can run the external programs in either of the following ways:

- Open an `Xterm` window. At the command prompt, type the name of the target executable, followed by possible options, such as:

  ```
  ./ext_example -tf inf -w
  ```

- Alternatively, you can launch the target executable from the MATLAB Command Window. You must run it in the background so that you can still access the Simulink environment. The command must be preceded by an exclamation point (`!`), dot slash (./ indicating the current directory), and followed by an ampersand (`&`), as in the following example:

  ```
  !./ext_example -tf inf -w &
  ```

  The ampersand (`&`) causes the operating system to spawn another process to run the target executable.

Command-Line Options for the External Program

External mode target executables generated by the Simulink Coder code generator support the following command-line options:

- `-tf n`

  The `-tf` option overrides the stop time set in the Simulink model. The argument `n` specifies the number of seconds the program will run. The value `inf` directs the model

to run indefinitely. In this case, the model code runs until the target program receives a stop message from the Simulink engine.

The following example sets the stop time to 10 seconds.

```
ext_example -tf 10
```

When integer-only ERT targets are built and executed in external mode, the stop time parameter (`-tf`) is interpreted by the target as the number of base rate ticks rather than the number of seconds to execute.

- `-w`

  Instructs the target program to enter a wait state until it receives a message from the host. At this point, the target is running, but not executing the model code. The start message is sent when you select **Start Real-Time Code** from the **Simulation** menu or click the **Start Real-Time Code** button in the External Mode Control Panel.

  Use the `-w` option if you want to view data from time step 0 of the target program execution, or if you want to modify parameters before the target program begins execution of model code.

- `-port n`

  Specifies the TCP/IP port number or the serial port ID, `n`, for the target program. The port number of the target program must match that of the host for TCP/IP transport. The port number depends on the type of transport.

  - For TCP/IP transport: Port number is an integer between `256` and `65535`, with the default value being `17725`.
  - For serial transport: Port ID is an integer or a character vector. For example, specify the port ID of a USB to serial converter as `'/dev/ttyusb0'`

- `-baud r`

  Specified as an integer, this option is only available for serial transport.

Implement an External Mode Protocol Layer

If you want to implement your own transport layer for external mode communication, you must modify certain code modules provided by the Simulink Coder product and create a new external interface MEX-file. See “Create a Transport Layer for External Communication” (Simulink Coder).

Use External Mode Programmatically

You can run external-mode applications from the MATLAB command line or programmatically in scripts. Use the `get_param` and `set_param` commands to retrieve and set the values of model simulation command-line parameters, such as `SimulationMode` and `SimulationCommand`, and external mode command-line parameters, such as `ExtModeCommand` and `ExtModeTrigType`.

The following model simulation commands assume that a Simulink model is open and that you have loaded a target program to which the model will connect using external mode.

1. Change the Simulink model to external mode:

   ```
   set_param(gcs,'SimulationMode','external')
   ```

2. Connect the open model to the loaded target program:

   ```
   set_param(gcs,'SimulationCommand','connect')
   ```

3. Start running the target program:

   ```
   set_param(gcs,'SimulationCommand','start')
   ```

4. Stop running the target program:

   ```
   set_param(gcs,'SimulationCommand','stop')
   ```

5. Disconnect the target program from the model:

   ```
   set_param(gcs,'SimulationCommand','disconnect')
   ```

To tune a workspace parameter, change its value at the command prompt. If the workspace parameter is a `Simulink.Parameter` object, assign the new value to the `Value` property.

```
myVariable = 5.23;
myParamObj.Value = 5.23;
```

To download the workspace parameter in external mode, you update the model diagram. The following model simulation command initiates a model update:

```
set_param(gcs,'SimulationCommand','update')
```

To trigger or cancel data uploading to scopes, use the `ExtModeCommand` values `armFloating` and `cancelFloating`, or `armWired` and `cancelWired`. For example, to trigger and then cancel data uploading to wired (nonfloating) scopes:

```
set_param(gcs,'ExtModeCommand','armWired')
set_param(gcs,'ExtModeCommand','cancelWired')
```

The next table lists external mode command-line parameters that you can use in `get_param` and `set_param` commands. The table provides brief descriptions, valid values (bold type highlights defaults), and a mapping to External Mode dialog box equivalents. For external mode parameters that are equivalent to **Interface** pane options in the Configuration Parameters dialog box, see "Model Configuration Parameters: Code Generation Interface" (Simulink Coder).

External Mode Command-Line Parameters

Parameter and Values	Dialog Box Equivalent	Description
`ExtModeAddSuffixToVar` **`off`**, on	Enable Data Archiving: **Append file suffix to variable names** check box	Increment variable names for each incremented filename.
`ExtModeArchiveDirName` *`character vector`*	Enable Data Archiving: **Directory** text field	Save data in specified folder.
`ExtModeArchiveFileName` *`character vector`*	Enable Data Archiving: **File** text field	Save data in specified file.
`ExtModeArchiveMode` *`character vector`* - **`off`**, on	Enable Data Archiving: **Enable archiving** check box	Activate automated data archiving features.
`ExtModeArmWhenConnect` `off`, **`on`**	External Signal & Triggering: **Arm when connecting to target** check box	Arm the trigger as soon as the Simulink Coder software connects to the target.
`ExtModeAutoIncOneShot` **`off`**, on	Enable Data Archiving: **Increment file after one-shot** check box	Save new data buffers in incremental files.
`ExtModeAutoUpdateStatusClock` (Microsoft Windows platforms only) `off`, **`on`**	Not available	Continuously upload and display target time on the model window status bar.
`ExtModeBatchMode` **`off`**, on	External Mode Control Panel: **Batch download** check box	Enable or disable downloading of parameters in batch mode.
`ExtModeChangesPending` **`off`**, on	Not available	When `ExtModeBatchMode` is enabled, indicates whether parameters remain in the queue of parameters

Parameter and Values	Dialog Box Equivalent	Description
		to be downloaded to the target.
`ExtModeCommand` *`character vector`* - `armFloating`, `armWired`, `cancelFloating`, `cancelWired`	• `armFloating` and `cancelFloating` are equivalent to selecting and clearing External Mode Control Panel check box **Floating scope** > **Enable data uploading** • `armWired` and `cancelWired` are equivalent to External Mode Control Panel buttons **Arm Trigger** and **Cancel Trigger**	Issue an external mode command to the target program.
`ExtModeConnected` **`off`**, `on`	External Mode Control Panel: **Connect/Disconnect** button	Indicate the state of the connection with the target program.
`ExtModeEnableFloating` `off`, **`on`**	External Mode Control Panel: **Enable data uploading** check box	Enable or disable the arming and canceling of triggers when a connection is established with floating scopes.
`ExtModeIncDirWhenArm` **`off`**, `on`	Enable Data Archiving: **Increment directory when trigger armed** check box	Write log files to incremental folders each time the trigger is armed.
`ExtModeLogAll` `off`, **`on`**	External Signal & Triggering: **Select all** check box	Upload available signals from the target to the host.
`ExtModeParamChangesPending` **`off`**, `on`	Not available	When the Simulink Coder software is connected to the target and `ExtModeBatchMode` is enabled, indicates whether parameters remain in the queue of parameters to be downloaded to the

Parameter and Values	Dialog Box Equivalent	Description
		target. More efficient than `ExtModeChangesPending`, because it checks for a connection to the target.
`ExtModeSkipDownloadWhenConnect` **`off`**, `on`	Not available	Connect to the target program without downloading parameters.
`ExtModeTrigDelay` *`integer`* (**0**)	External Signal & Triggering: **Delay** text field	Specify the amount of time (expressed in base rate steps) that elapses between a trigger occurrence and the start of data collection.
`ExtModeTrigDirection` *`character vector`* - **`rising`**, `falling`, `either`	External Signal & Triggering: **Direction** menu	Specify the direction in which the signal must be traveling when it crosses the threshold value.
`ExtModeTrigDuration` *`integer`* (**1000**)	External Signal & Triggering: **Duration** text field	Specify the number of base rate steps for which external mode is to log data after a trigger event.
`ExtModeTrigDurationFloating` *`character vector`* - *`integer`* (**`auto`**)	External Mode Control Panel: **Duration** text field	Specify the duration for floating scopes. If `auto` is specified, the value of `ExtModeTrigDuration` is used.
`ExtModeTrigElement` *`character vector`* - *`integer`*, **`any`**, `last`	External Signal & Triggering: **Element** text field	Specify the elements of the input port of the specified trigger block that can cause the trigger to fire.
`ExtModeTrigHoldOff` *`integer`* (**0**)	External Signal & Triggering: **Hold-off** text field	Specify the base rate steps between when a trigger event terminates and the trigger is rearmed.

Parameter and Values	Dialog Box Equivalent	Description
`ExtModeTrigLevel` *integer* (**0**)	External Signal & Triggering: **Level** text field	Specify the threshold value the trigger signal must cross to fire the trigger.
`ExtModeTrigMode` *character vector* - **`normal`**, `oneshot`	External Signal & Triggering: **Mode** menu	Specify whether the trigger is to rearm automatically after each trigger event or whether only one buffer of data is to be collected each time the trigger is armed.
`ExtModeTrigPort` *character vector* - *integer* (**1**), `last`	External Signal & Triggering: **Port** text field	Specify the input port of the specified trigger block for which elements can cause the trigger to fire.
`ExtModeTrigType` *character vector* - **`manual`**, `signal`	External Signal & Triggering: **Source** menu	Specify whether to start logging data when the trigger is armed or when a specified trigger signal satisfies trigger conditions.
`ExtModeUploadStatus` *character vector* - `inactive`, `armed`, `uploading`	Not available	Return the status of the external mode upload mechanism — inactive, armed, or uploading.
`ExtModeWriteAllDataToWs` **`off`**, `on`	Enable Data Archiving: **Write intermediate results to workspace** check box	Write intermediate results to the workspace.

Animate Stateflow Charts in External Mode

If you have Stateflow, you can animate a chart in external mode. In external mode, you can animate states in a chart, and view test point signals in a floating scope or signal viewer.

- "Animate States During Simulation in External Mode" on page 41-54
- "View Test Point Data in Floating Scopes and Signal Viewers" on page 41-54

Animate States During Simulation in External Mode

To animate states in a chart in external mode:

1. Load the chart you want to animate to the target machine.
2. Open the Model Configuration Parameters dialog box.
3. In the left Select pane, select **Code Generation > Interface**.
4. In the **Data exchange interface** section, select **External mode** and click **OK**.
5. In the Simulink Editor, select **Code > External Mode Control Panel**.
6. In the External Mode Control Panel dialog box, click **Signal & Triggering**.
7. In the External Signal & Triggering dialog box, set these parameters.

In:	Select:
Signal selection pane	Chart you want to animate
Trigger pane	**Arm when connecting to target** check box
Trigger pane	`normal` from drop-down menu in **Mode** field

8. Build the model to generate an executable file.
9. Start the target in the background. At the MATLAB prompt, type:

   ```
   !model_name.exe -w &
   ```

   For example, if the name of your model is `my_control_sys`, enter this command:

   ```
   !my_control_sys.exe -w &
   ```

   `-w` allows the target code to wait for the Simulink model connection.
10. In the Model Editor, select **Simulation > Mode > External**, and then select **Simulation > Connect to Target**.
11. Start simulation. The chart highlights states as they execute.

View Test Point Data in Floating Scopes and Signal Viewers

When you simulate a chart in external mode, you can designate chart data of local scope to be test points and view the test point data in floating scopes and signal viewers.

To view test point data during simulation in external mode:

1. Open the Model Explorer and for each data you want to view, follow these steps:

   a. In the middle **Contents** pane, select the state or local data of interest.
   b. In the right **Dialog** pane, select the **Logging** tab and select **Test point** check box.
2. From a floating scope or signal viewer, click the signal selection button:

   The Signal Selector dialog box opens.
3. In the Signal Selector **Model hierarchy** pane, select the chart.
4. In the Signal Selector **List contents** menu, select **Testpointed/Logged signals only** and then select the signals you want to view.
5. Simulate the model in external mode as described in "Animate States During Simulation in External Mode" on page 41-54.

   The scope or viewer displays the values of the test point signals as the simulation runs.

For more information, see "Behavior of Scopes and Viewers with Rapid Accelerator Mode" (Simulink).

External Mode Limitations

Changing Parameters

In general, you cannot change a parameter if doing so results in a change in the structure of the model. For example, you cannot change

- The number of states, inputs, or outputs of a block
- The sample time or the number of sample times
- The integration algorithm for continuous systems
- The name of the model or of a block
- The parameters to the Fcn block

If you make these changes to the block diagram, then you must rebuild the program with newly generated code.

You can change parameters in transfer function and state space representation blocks in specific ways:

- The parameters (numerator and denominator polynomials) for the Transfer Fcn (continuous and discrete) and Discrete Filter blocks can be changed (as long as the number of states does not change).
- Zero entries in the State-Space and Zero Pole (both continuous and discrete) blocks in the user-specified or computed parameters (that is, the A, B, C, and D matrices obtained by a zero-pole to state-space transformation) cannot be changed once external simulation is started.
- In the State-Space block, if you specify the matrices in the controllable canonical realization, then all changes to the A, B, C, D matrices that preserve this realization and the dimensions of the matrices are allowed.

If the Simulink block diagram does not match the external program, the Simulink engine displays an error informing you that the checksums do not match (that is, the model has changed since you generated code). This means that you must rebuild the program from the new block diagram (or reload another one) to use external mode.

If the external program is not running, the Simulink engine displays an error informing you that it cannot connect to the external program.

Mixing 32-Bit and 64-Bit Architectures

When you use external mode, the machine running the Simulink product and the machine running the target executable must have matching bit architectures, either 32-

bit or 64-bit. The Simulink Coder software varies a model's checksum based on whether it is configured for a 32-bit or 64-bit platform.

If you attempt to connect from a 32-bit machine to a 64-bit machine or vice versa, the external mode connection fails.

Uploading Data

External mode does not support uploading data values for fixed-point or enumerated types into workspace parameters.

Uploading Variable-Size Signals

External mode does not support uploading variable-size signals for the following targets:

- Simulink Real-Time
- Texas Instruments™ C2000™

Signal Value Display in Simulation

External mode does not support graphical display of signal values in models (described in "Displaying Signal Values in Model Diagrams" (Simulink)). For example, you cannot use the **Data Display in Simulation** menu selections **Show Value Labels When Hovering**, **Toggle Value Labels When Clicked**, and **Show Value Label of Selected Port**.

Tunable Structure Parameters

External mode does not support uploading or downloading tunable structure parameters.

Archiving Data

External mode supports the Scope and To Workspace blocks for archiving data to disk. However, external mode does not support scopes other than the Scope block for archiving data. For example, you cannot use Floating Scope blocks or Signal and Scope Manager viewer objects to archive data in external mode.

Scopes in Referenced Models

In a model hierarchy, if the top model simulates in external mode and a referenced model simulates in normal or accelerator mode, scopes in the referenced model are not displayed.

However, if the top model is changed to simulate in normal mode, the behavior of scopes in the referenced models differs between normal and accelerator mode. Scopes in a referenced model simulating in normal mode are displayed, while scopes in a referenced model simulating in accelerator mode are not displayed.

Simulation Start Time

External mode does not support nonzero simulation start times. In the Configuration Parameters dialog box, **Solver** pane, leave **Start time** set to the default value of `0.0`.

File-Scoped Data

External mode does not support file-scoped data, for example, data items to which you apply the built-in custom storage class `FileScope`. File-scoped data are not externally accessible.

Use of `printf` Statements

External mode simulations support the use of `printf` calls to display error and information messages from the target program. For some target hardware, the use of `printf` statements can increase the external mode binary file size. To disable `printf` calls, specify the preprocessor macro definition `EXTMODE_DISABLEPRINTF` for your target program compiler.

Command-Line Arguments

External mode simulations support the use of command-line arguments for running target programs. These limitations apply:

- Parsing of the command-line arguments requires the `sscanf` function, which increases the program size for some target hardware.
- Some target programs do not accept command-line arguments.

To disable the processing of command-line arguments, specify the preprocessor macro definition `EXTMODE_DISABLE_ARGS_PROCESSING=1` for your target program compiler.

Related Examples

- “Use External Mode with the ERT Target” on page 54-5

42

Logging in Simulink Coder

Log Program Execution Results

Multiple techniques are available by which a program generated by the Simulink Coder software can save data to a MAT-file for analysis. A generated executable can save system states, outputs, and simulation time at each model execution time step. The data is written to a MAT-file, named (by default) *model*`.mat`, where *model* is the name of your model. See "Log Data for Analysis" on page 42-2 for a data logging tutorial.

Note: Data logging is available only for system target files that have access to a file system. In addition, only the RSim target executables are capable of accessing MATLAB workspace data.

For MAT-file logging limitations, see the configuration parameter "MAT-file logging" (Simulink Coder).

In this section...
"Log Data for Analysis" on page 42-2
"Configure State, Time, and Output Logging" on page 42-9
"Log Data with Scope and To Workspace Blocks" on page 42-11
"Log Data with To File Blocks" on page 42-11
"Data Logging Differences Between Single- and Multitasking" on page 42-12

Log Data for Analysis

- "Set Up and Configure Model" on page 42-2
- "Data Logging During Simulation" on page 42-4
- "Data Logging from Generated Code" on page 42-7

Set Up and Configure Model

This example shows how data generated by a copy of the model `slexAircraftExample` is logged to the file `myAircraftExample.mat`. Refer to "Build Process Workflow for a Real-Time STF" on page 40-30 for instructions on setting up a copy of `slexAircraftExample` as`myAircraftExample` in a working folder if you have not done so already.

Note: When you configure the code generator to produce code that includes support for data logging during execution, the code generator can include text for block names in the block paths included in the log file. If the text includes characters that are unrepresented in the character set encoding for the model, the code generator replaces the characters with XML escape sequences. For example, the code generator replaces the Japanese full-width Katakana letter ｱ with the escape sequence `&#x30A2;`. For more information, see "Internationalization and Code Generation" (Simulink Coder).

To configure data logging, open the Configuration Parameters dialog box and select the **Data Import/Export** pane. The process is the same as configuring a Simulink model to save output to the MATLAB workspace. For each workspace return variable you define and enable, the Simulink Coder software defines a parallel MAT-file variable. For example, if you save simulation time to the variable `tout`, your generated program logs the same data to a variable named `rt_tout`. You can change the prefix `rt_` to a suffix (`_rt`), or eliminate it entirely. You do this by setting **Configuration Parameters** > **All Parameters** > **MAT-file variable name modifier**.

Simulink lets you log signal data from anywhere in a model. In the Simulink Editor, select the signals that you want to log and then in the **Simulation Data Inspector** button drop-down, select **Log Selected Signals**. However, the Simulink Coder software does not use this method of signal logging in generated code. To log signals in generated code, you must either use the **Data Import/Export** options described below or include To File or To Workspace blocks in your model.

Note: If you enable MAT-file and signal logging (through the **Data Import/Export** pane) and select signals for logging (through the Simulink Editor), you see the following warning when you build the model:

```
Warning: MAT-file logging does not support signal logging.
When your model code executes, the signal logging variable 'rt_logsout' will
not be saved to the MAT-file.
```

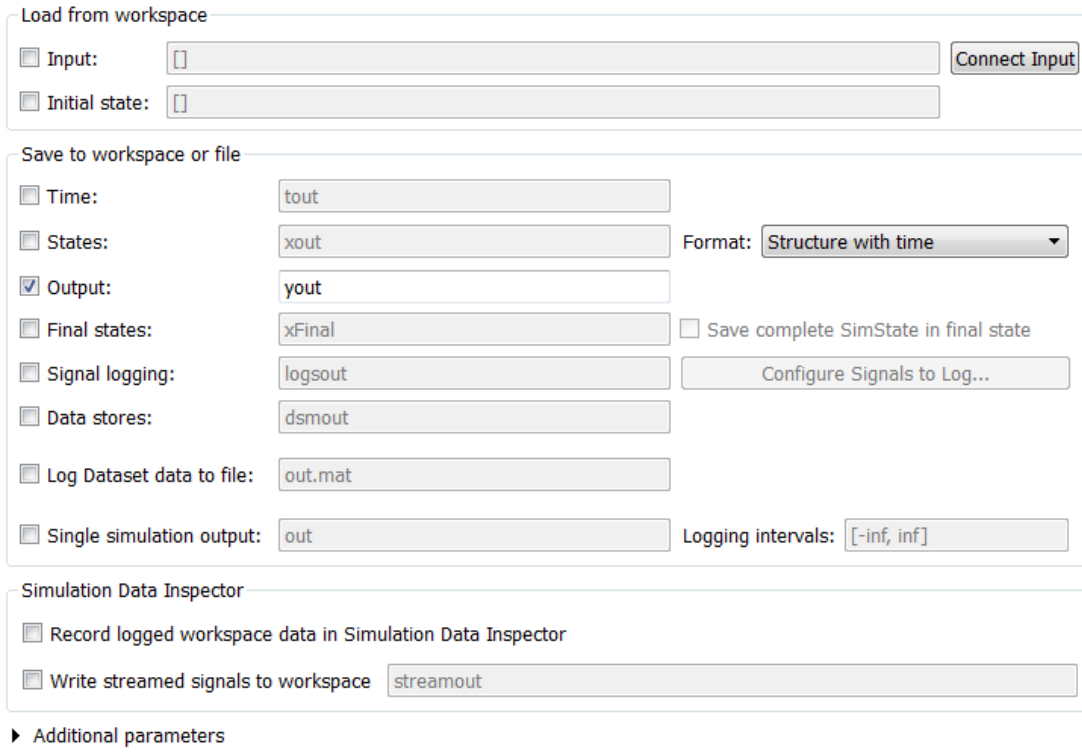
To avoid this warning, clear the **Data Import/Export** > **Signal logging** check box.

In this example, you modify the `myAircraftExample` model so that the generated program saves the simulation time and system outputs to the file `myAircraftExample.mat`. Then you load the data into the base workspace and plot simulation time against one of the outputs. The `myAircraftExample` model should be configured as described in "Build Process Workflow for a Real-Time STF" on page 40-30.

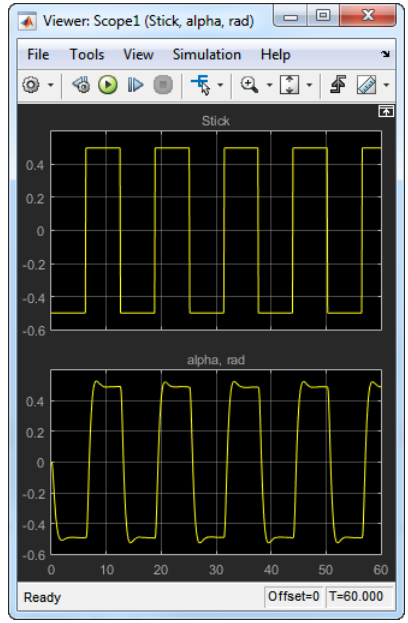
Data Logging During Simulation

To use the data logging feature:

1. Open the `myAircraftExample` model if it is not already open.
2. Open the Configuration Parameters dialog box by selecting **Simulation** > **Model Configuration Parameters** from the model window.
3. Select the **Data Import/Export** pane. The **Data Import/Export** pane lets you specify which outport data is to be saved to the workspace and what variable names to use for it.
4. Set **Format** to `Structure with time`. When you select this format, Simulink saves the model states and outputs in structures that have their names specified in the **Save to workspace or file** area. By default, the structures are `xout` for states and `yout` for output. The structure used to save output has two top-level fields: `time` and `signals`. The `time` field contains a vector of simulation times and `signals` contains an array of substructures, each of which corresponds to a model output port.
5. Select the **Output** option. This tells Simulink to save output signal data during simulation as a variable named `yout`. Selecting **Output** enables the code generator to create code that logs the root Output block (`alpha, rad`) to a MAT-file.
6. Set **Decimation** to `1`.
7. If other options are enabled, clear them. The figure below shows how the dialog box should appear.



8. Click **Apply** and **OK** to register your changes and close the dialog box.
9. Save the model.
10. In the model window, double-click the scope symbol next to the Aircraft Dynamics Model block, then run the model by choosing **Simulation** > **Run** in the model window. The resulting scope display is shown below.



11 Verify that the simulation time and outputs have been saved to the base workspace in MAT-files. At the MATLAB prompt, type:

```
whos yout
```

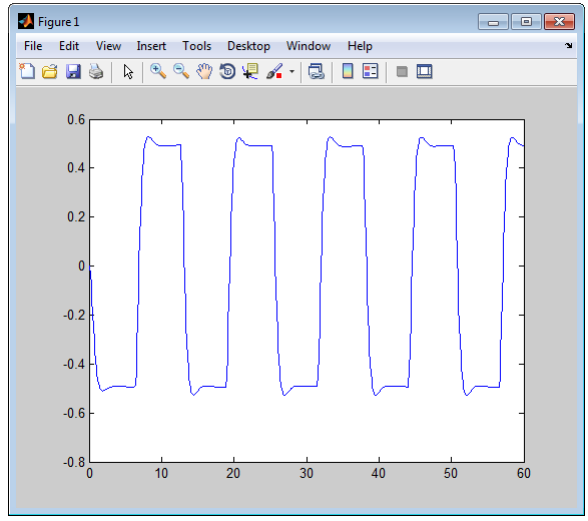
Simulink displays:

```
  Name        Size            Bytes  Class     Attributes

  yout        1x1             10756  struct
```

12 Verify that `alpha, rad` was logged by plotting simulation time versus that variable. In the Command Window, type:

```
plot(yout.time,yout.signals.values)
```

The resulting plot is shown below.



Data Logging from Generated Code

In the second part of this example, you build and run a Simulink Coder executable of the `myAircraftExample` model that outputs a MAT-file containing the simulation time and output you previously examined. Even though you have already generated code for the `myAircraftExample` model, you must now regenerate that code because you have changed the model by enabling data logging. The steps below explain this procedure.

To avoid overwriting workspace data with data from simulation runs, the code generator modifies identifiers for variables logged by Simulink. You can control these modifications.

1. Set **Configuration Parameters > All Parameters > MAT-file variable name modifier** to `_rt`. This adds the suffix `_rt` to each variable that you selected to be logged in the first part of this example.
2. Click **Apply** and **OK** to register your changes and close the dialog box.
3. Save the model.
4. Build an executable.
5. When the build concludes, run the executable with the command:

   ```
   !myAircraftExample
   ```
6. The program now produces two message lines, indicating that the MAT-file has been written.

```
** starting the model **
** created myAircraftExample.mat **
```

7 Load the MAT-file data created by the executable and look at the workspace variables from simulation and the generated program by typing:

```
load myAircraftExample.mat
whos yout*
```

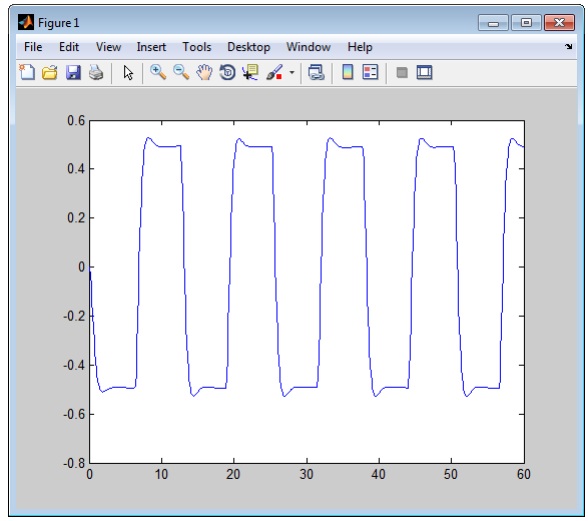
Simulink displays:

```
  Name           Size            Bytes  Class     Attributes

  yout           1x1             10756  struct
  yout_rt        1x1             10756  struct
```

Note the size and bytes of the structures resulting from the simulation run and generated code are the same.

8 Plot the generated code output by entering the following command in the Command Window:

```
plot(yout_rt.time,yout_rt.signals.values)
```



The plot should be identical to the plot that you produced in the previous part of this example.

Tip: For UNIX platforms, run the executable in the Command Window with the syntax `!./executable_name`. If preferred, run the executable from an OS shell with the syntax `./executable_name`. For more information, see "Run External Commands, Scripts, and Programs" (MATLAB).

Configure State, Time, and Output Logging

The **Data Import/Export** pane enables a generated program to save system states, outputs, and simulation time at each model execution time step. The data is written to a MAT-file, named (by default) `model.mat`.

Before using this data logging feature, you should learn how to configure a Simulink model to return output to the MATLAB workspace. This is discussed in "Export Simulation Data" (Simulink).

For each workspace return variable that you define and enable, the code generator defines a MAT-file variable. For example, if your model saves simulation time to the workspace variable `tout`, your generated program logs the same data to a variable named (by default) `rt_tout`.

The code generated by the code generator logs the following data:

- Root Outport blocks

  The default MAT-file variable name for system outputs is `rt_yout`.

  The sort order of the `rt_yout` array is based on the port number of the Outport block, starting with 1.
- Continuous and discrete states in the model

  The default MAT-file variable name for system states is `rt_xout`.
- Simulation time

  The default MAT-file variable name for simulation time is `rt_tout`.

- "Override Default MAT-File Variable Names" on page 42-10

- “Override Default MAT-File Name or Buffer Size” on page 42-10

Override Default MAT-File Variable Names

By default, the code generation software prefixes the text `rt_` to the variable names for system outputs, states, and simulation time to form MAT-file variable names. To change this prefix for a model, select a prefix (`rt_`), a suffix (`_rt`), or no modifier (`none`) for **Configuration Parameters > All Parameters > MAT-file variable name modifier**. Other system target files might not support this parameter.

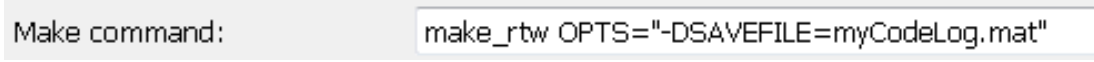
Override Default MAT-File Name or Buffer Size

You can specify compiler options to override the following MAT-file attributes in generated code:

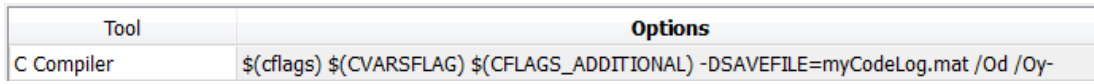
MAT-File Attribute	Default	Compiler Option
Name	*`model`*`.mat`	`-DSAVEFILE=`*`filename`*
Size of data logging buffer	1024 bytes	`-DDEFAULT_BUFFER_SIZE=`*`n`*

Note: Valid option syntax can vary among compilers. For example, Microsoft Visual C++ compilers typically accept `/DSAVEFILE=`*`filename`* as well as `-DSAVEFILE=`*`filename`*.

For a template makefile (TMF) based target, append the compiler option to the **Make command** field on the **Code Generation** pane of the Configuration Parameters dialog box. For example:



For a toolchain-based target such as GRT or ERT, add the compiler option to the **Build configuration** settings on the **Code Generation** pane of the Configuration Parameters dialog box. Set **Build configuration** to `Specify`, and add the compiler option to the **C Compiler** row of the **Tool/Options** table. For example:



To add the compiler option to a custom toolchain, you can modify and reregister the custom toolchain using the procedures shown in the example “Adding a Custom

Toolchain" (MATLAB Coder). For example, to add the compiler option to the MATLAB source file for the custom toolchain, you could define `myCompilerOpts` as follows:

```
optimsOffOpts    = {'/c /Od'};
optimsOnOpts     = {'/c /O2'};
cCompilerOpts    = '$(cflags) $(CVARSFLAG) $(CFLAGS_ADDITIONAL)';
cppCompilerOpts  = '$(cflags) $(CVARSFLAG) $(CPPFLAGS_ADDITIONAL)';
myCompilerOpts   = {' -DSAVEFILE=myCodeLog.mat '};
...
```

Then you can add `myCompilerOpts` to the flags for each configuration and compiler to which it applies, for example:

```
cfg = tc.getBuildConfiguration('Faster Builds');
cfg.setOption('C Compiler', horzcat(cCompilerOpts, myCompilerOpts, optimsOffOpts));
```

As shown in "Adding a Custom Toolchain" (MATLAB Coder), after modifying the custom toolchain, you save the configuration to a MAT-file and refresh the target registry.

Log Data with Scope and To Workspace Blocks

The code generated by the code generator also logs data from these sources:

- Scope blocks that have the **Log data to workspace** parameter enabled

  You must specify the variable name and data format in each Scope block's dialog box.
- To Workspace blocks in the model

  You must specify the variable name and data format in each To Workspace block's dialog box.

The variables are written to `model.mat`, along with variables logged from the **Workspace I/O** pane.

Log Data with To File Blocks

You can also log data to a To File block. The generated program creates a separate MAT-file (distinct from `model.mat`) for each To File block in the model. The file contains the block time and input variable(s). You must specify the filename, variable names, decimation, and sample time in the To File block dialog box.

Note: Models referenced by Model blocks do not perform data logging in that context except for states, which you can include in the state logged for top models. Code

generated by the Simulink Coder software for referenced models does not perform data logging to MAT-files.

Data Logging Differences Between Single- and Multitasking

When logging data in single-tasking and multitasking systems, you will notice differences in the logging of

- Noncontinuous root Outport blocks
- Discrete states

In multitasking mode, the logging of states and outputs is done after the first task execution (and not at the end of the first time step). In single-tasking mode, the code generated by the build procedure logs states and outputs after the first time step.

See Data Logging in Single-Tasking and Multitasking Model Execution (Simulink Coder) for more details on the differences between single-tasking and multitasking data logging.

Note: The rapid simulation target (RSim) provides enhanced logging options. See "Accelerate, Refine, and Test Hybrid Dynamic System on Host Computer by Using RSim System Target File" (Simulink Coder) for more information.

Data Interchange Using the C API in Simulink Coder

Exchange Data Between Generated and External Code Using C API

Some Simulink Coder applications must interact with signals, states, root-level inputs/outputs, or parameters in the generated code for a model. For example, calibration applications monitor and modify parameters. Signal monitoring or data logging applications interface with signal, state, and root-level input/output data. Using the Simulink Coder C API, you can build target applications that log signals, states, and root-level inputs/outputs, monitor signals, states, and root-level inputs/outputs, and tune parameters, while the generated code executes.

The C API minimizes its memory footprint by sharing information common to signals, states, root-level inputs/outputs, and parameters in smaller structures. Signal, state, root-level input/output, and parameter structures include an index into the structure map, allowing multiple signals, states, root-level inputs/outputs, or parameters to share data.

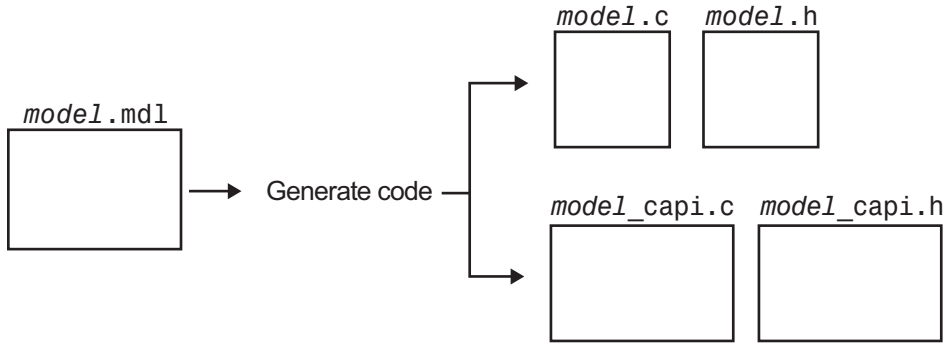
To get started with an example, see "Use C API to Access Model Signals and States" on page 43-24 or "Use C API to Access Model Parameters" on page 43-30.

In this section...

Generated C API Files

When you configure a model to use the C API, the Simulink Coder code generator generates two additional files, *`model`*`_capi.c` (or `.cpp`) and *`model`*`_capi.h`, where *`model`* is the name of the model. The code generator places the two C API files in the build folder, based on settings in the Configuration Parameters dialog box. The C API source code file contains information about global block output signals, states, root-level inputs/outputs, and global parameters defined in the generated code model source code. The C API header file is an interface header file between the model source code and

the generated C API. You can use the information in these C API files to create your application. Among the files generated are those shown in the next figure.



Generated Files with C API Selected

Note: When you configure the code generator to produce code that includes support for the C API interface and data logging, the code generator can include text for block names in the block paths logged to C API files *model*`_capi.c` (or `.cpp`) and *model*`_capi.h`. If the text includes characters that are unrepresented in the character set encoding for the model, the code generator replaces the characters with XML escape sequences. For example, the code generator replaces the Japanese full-width Katakana letter ア with the escape sequence `&#x30A2;`. For more information, see "Internationalization and Code Generation" (Simulink Coder).

Generate C API Files

To generate C API files for your model:

1. Select the C API interface for your model. There are two ways to select the C API interface for your model, as described in the following sections.
   - "Select C API with Configuration Parameters Dialog" on page 43-4
   - "Select C API from the Command Line" on page 43-4
2. Generate code for your model.

After generating code, you can examine the files *model*`_capi.c` (or `.cpp`) and *model*`_capi.h` in the model build folder.

Select C API with Configuration Parameters Dialog

1. Open your model, and open the Configuration Parameters dialog box.
2. Go to the **Code Generation** > **Interface** pane and, in the **Data exchange interface** subgroup, select one or more `C API` options. Based on the options you select, support for accessing signals, parameters, states, and root-level I/O will appear in the C API generated code.

   - If you want to generate C API code for global block output signals, select **Generate C API for: signals**.
   - If you want to generate C API code for global block parameters, select **Generate C API for: parameters**.
   - If you want to generate C API code for discrete and continuous states, select **Generate C API for: states**.
   - If you want to generate C API code for root-level inputs and outputs, select **Generate C API for: root-level I/O**.

Select C API from the Command Line

From the MATLAB command line, you can use the `set_param` function to select or clear the C API check boxes on the **Interface** pane of the Configuration Parameters dialog box. At the MATLAB command line, enter one or more of the following commands, where `modelname` is the name of your model.

To select **Generate C API for: signals**, enter:

```
set_param('modelname','RTWCAPISignals','on')
```

To clear **Generate C API for: signals**, enter:

```
set_param('modelname','RTWCAPISignals','off')
```

To select **Generate C API for: parameters**, enter:

```
set_param('modelname','RTWCAPIParams','on')
```

To clear **Generate C API for: parameters**, enter:

```
set_param('modelname','RTWCAPIParams','off')
```

To select **Generate C API for: states**, enter:

```
set_param('modelname','RTWCAPIStates','on')
```

To clear **Generate C API for: states**, enter:

```
set_param('modelname','RTWCAPIStates','off')
```

To select **Generate C API for: root-level I/O**, enter:

```
set_param('modelname','RTWCAPIRootIO','on')
```

To clear **Generate C API for: root-level I/O**, enter:

```
set_param('modelname','RTWCAPIRootIO','off')
```

Description of C API Files

About C API Files

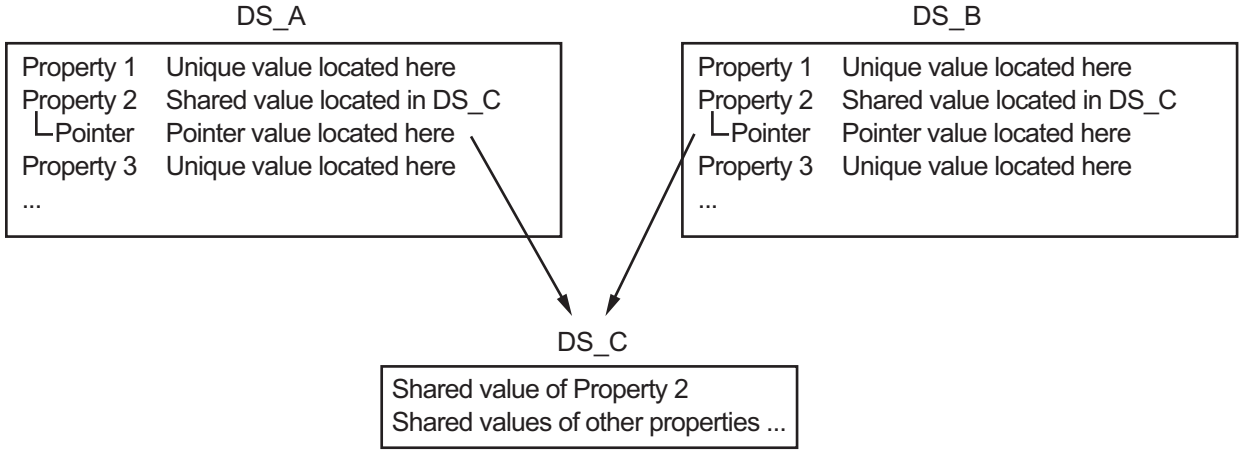
The *`model`*`_capi.c` (or `.cpp`) file provides external applications with a consistent interface to model data. Depending on your configuration settings, the data could be a signal, state, root-level input or output, or parameter. In this document, the term *data item* refers to either a signal, a state, a root-level input or output, or a parameter. The C API uses structures that provide an interface to the data item properties. The interface packages the properties of each data item in a data structure. If the model contains multiple data items, the interface generates an array of data structures. The members of a data structure map to data properties.

To interface with data items, an application requires the following properties for each data item:

- Name

- Block path
- Port number (for signals and root-level inputs/outputs only)
- Address
- Data type information: native data type, data size, complexity, and other attributes
- Dimensions information: number of rows, number of columns, and data orientation (scalar, vector, matrix, or *n*-dimensional)
- Fixed-point information: slope, bias, scale type, word length, exponent, and other attributes
- Sample-time information (for signals, states, and root-level inputs/outputs only): sample time, task identifier, frames

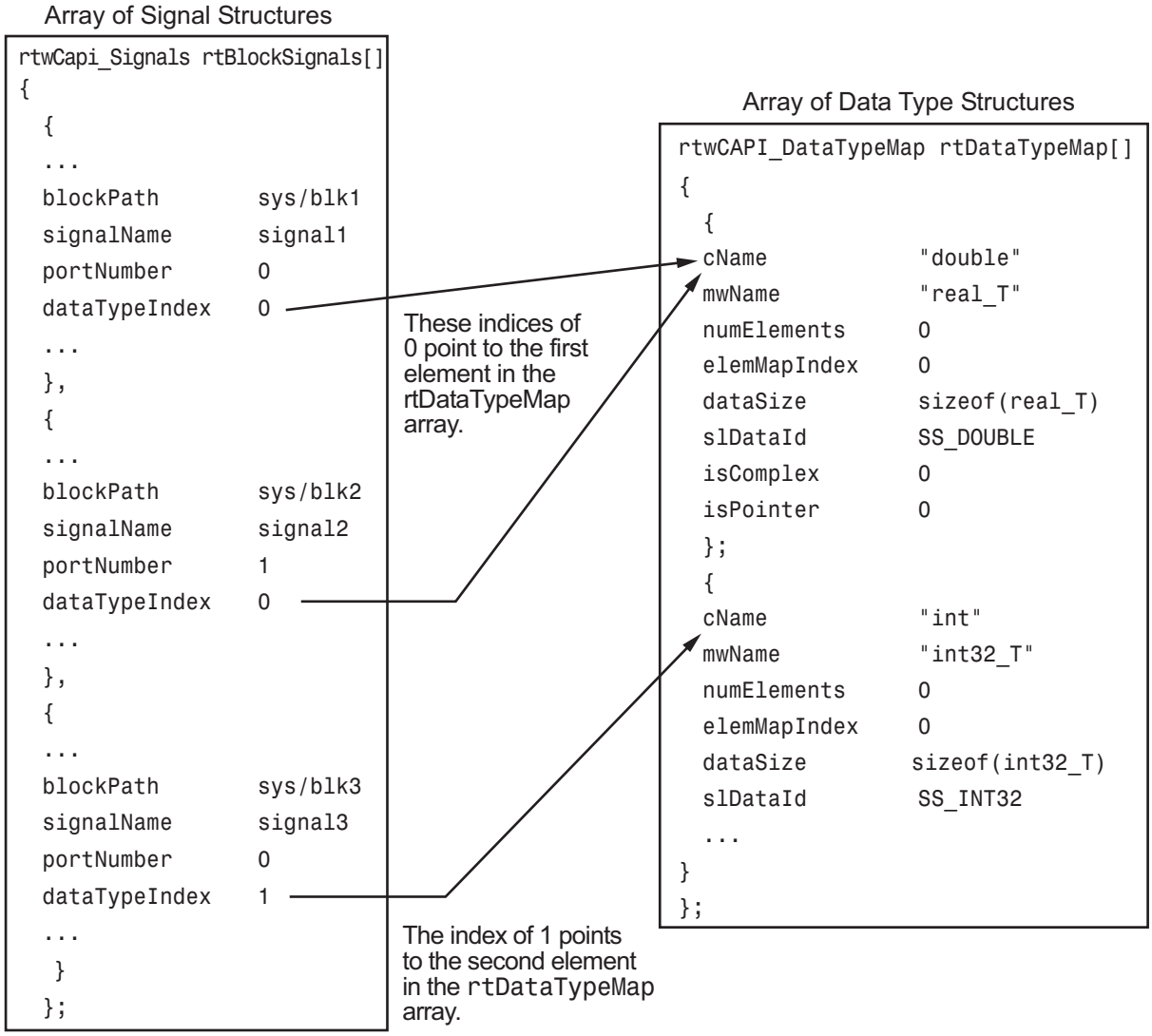
As illustrated in the next figure, the properties of data item A, for example, are located in data structure DS_A. The properties of data item B are located in data structure DS_B.



Some property *values* can be unique to each data item, and there are some property values that several data items can share in common. Name, for example, has a unique value for each data item. The interface places the unique property values directly in the structure for the data item. The name value of data item A is in DS_A, and the name value of data item B is in DS_B.

But data type could be a property whose value several data items have in common. The ability of some data items to share a property allows the C API to have a reuse feature. In this case, the interface places only an index value in DS_A and an index value in

DS_B. These indices point to a different data structure, DS_C, that contains the actual data type value. The next figure shows this scheme with more detail.



The figure shows three signals. `signal1` and `signal2` share the same data type, `double`. Instead of specifying this data type value in each signal data structure, the

interface provides only an index value, 0, in the structure. `"double"` is described by entry 0 in the `rtDataTypeMap` array, which is referenced by both signals. Additionally, property values can be shared between signals, states, root-level inputs/outputs, and parameters, so states, root-level inputs/outputs, and parameters also might reference the `double` entry in the `rtDataTypeMap` array. This reuse of information reduces the memory size of the generated interface.

Structure Arrays Generated in C API Files

As with data type, the interface maps other common properties (such as address, dimension, fixed-point scaling, and sample time) into separate structures and provides an index in the structure for the data item. For a complete list of structure definitions, refer to the file *`matlabroot`*`/rtw/c/src/rtw_capi.h`. This file also describes each member in a structure. The structure arrays generated in the *`model`*`_capi.c` (or `.cpp`) file are of structure types defined in the `rtw_capi.h` file. Here is a brief description of the structure arrays generated in *`model`*`_capi.c` (or `.cpp`):

- **`rtBlockSignals`** is an array of structures that contains information about global block output signals in the model. Each element in the array is of type `struct rtwCAPI_Signals`. The members of this structure provide the signal name, block path, block port number, address, and indices to the data type, dimension, fixed-point, and sample-time structure arrays.
- **`rtBlockParameters`** is an array of structures that contains information about the tunable block parameters in the model by block name and parameter name. Each element in the array is of type `struct rtwCAPI_BlockParameters`. The members of this structure provide the parameter name, block path, address, and indices to data type, dimension, and fixed-point structure arrays.
- **`rtBlockStates`** is an array of structures that contains information about discrete and continuous states in the model. Each element in the array is of type `struct rtwCAPI_States`. The members of this structure provide the state name, block path, type (continuous or discrete), and indices to the address, data type, dimension, fixed-point, and sample-time structure arrays.
- **`rtRootInputs`** is an array of structures that contains information about root-level inputs in the model. Each element in the array is of type `struct rtwCAPI_Signals`. The members of this structure provide the root-level input name, block path, block port number, address, and indices to the data type, dimension, fixed-point, and sample-time structure arrays.
- **`rtRootOutputs`** is an array of structures that contains information about root-level outputs in the model. Each element in the array is of type `struct`

`rtwCAPI_Signals`. The members of this structure provide the root-level output name, block path, block port number, address, and indices to the data type, dimension, fixed-point, and sample-time structure arrays.

- **`rtModelParameters`** is an array of structures that contains information about workplace variables that one or more blocks or Stateflow charts in the model reference as block parameters. Each element in the array is of data type `rtwCAPI_ModelParameters`. The members of this structure provide the variable name, address, and indices to data type, dimension, and fixed-point structure arrays.
- **`rtDataAddrMap`** is an array of base addresses of signals, states, root-level inputs/outputs, and parameters that appear in the `rtBlockSignals`, `rtBlockParameters`, `rtBlockStates`, and `rtModelParameters` arrays. Each element of the `rtDataAddrMap` array is a pointer to `void (void*)`.
- **`rtDataTypeMap`** is an array of structures that contains information about the various data types in the model. Each element of this array is of type `struct rtwCAPI_DataTypeMap`. The members of this structure provide the data type name, size of the data type, and information on whether or not the data is complex.
- **`rtDimensionMap`** is an array of structures that contains information about the various data dimensions in the model. Each element of this array is of type `struct rtwCAPI_DimensionMap`. The members of this structure provide information on the number of dimensions in the data, the orientation of the data (whether it is scalar, vector, or a matrix), and the actual dimensions of the data.
- **`rtFixPtMap`** is an array of structures that contains fixed-point information about the signals, states, root-level inputs/outputs, and parameters. Each element of this array is of type `struct rtwCAPI_FixPtMap`. The members of this structure provide information about the data scaling, bias, exponent, and whether or not the fixed-point data is signed. If the model does not have fixed-point data (signal, state, root-level input/output, or parameter), the Simulink Coder software assigns `NULL` or zero values to the elements of the `rtFixPtMap` array.
- **`rtSampleTimeMap`** is an array of structures that contains sampling information about the global signals, states, and root-level inputs/outputs in the model. (This array does not contain information about parameters.) Each element of this array is of type `struct rtwCAPI_SampleTimeMap`. The members of this structure provide information about the sample period, offset, and whether or not the data is frame-based or sample-based.

Generate Example C API Files

Subtopics “C API Signals” on page 43-11, “C API States” on page 43-14, “C API Root-Level Inputs and Outputs” on page 43-15, and “C API Parameters” on page

43-16 discuss generated C API structures using the example model `rtwdemo_capi`. To generate code from the example model, do the following:

1. Open the model by clicking the `rtwdemo_capi` link above or by typing `rtwdemo_capi` on the MATLAB command line.
2. If you want to generate C API structures for root-level inputs/outputs in `rtwdemo_capi`, open the Configuration Parameters dialog box, go to the **Code Generation** > **Interface** pane, and select **Generate C API for: root-level I/O**.

   Note: The setting of **Generate C API for: root-level I/O** must match between the top model and the referenced model. If you modify the option, save the top model and the referenced model to the same writable work folder.
3. Generate code for the model by double-clicking **Generate Code Using Simulink Coder**.

Note: The C API code examples in the next subtopics are generated with C as the target language.

This model has three global block output signals that will appear in C API generated code:

- `top_sig1`, which is a test point at the output of the Gain1 block in the top model
- `sig2_eg`, which appears in the top model and is defined in the base workspace as a `Simulink.Signal` object having storage class `ExportedGlobal`
- `bot_sig1`, which appears in the referenced model `rtwdemo_capi_bot` and is defined as a `Simulink.Signal` object having storage class `SimulinkGlobal`

The model also has two discrete states that will appear in the C API generated code:

- `top_state`, which is defined for the Delay1 block in the top model
- `bot_state`, which is defined for the Discrete Filter block in the referenced model

The model has root-level inputs/outputs that will appear in the C API generated code if you select the option **Generate C API for: root-level I/O**:

- Four root-level inputs, `In1` through `In4`
- Six root-level outputs, `Out1` through `Out6`

Additionally, the model has five global block parameters that will appear in C API generated code:

- `Kp` (top model `Gain1` block and referenced model `Gain2` block share)
- `Ki` (referenced model `Gain3` block)
- `p1` (lookup table `lu1d`)
- `p2` (lookup table `lu2d`)
- `p3` (lookup table `lu3d`)

C API Signals

The `rtwCAPI_Signals` structure captures signal information including the signal name, address, block path, output port number, data type information, dimensions information, fixed-point information, and sample-time information.

Here is the section of code in `rtwdemo_capi_capi.c` that provides information on C API signals for the top model in `rtwdemo_capi`:

```
/* Block output signal information */
static const rtwCAPI_Signals rtBlockSignals[] = {
  /* addrMapIndex, sysNum, blockPath,
   * signalName, portNumber, dataTypeIndex, dimIndex, fxpIndex, sTimeIndex
   */
  { 0, 0, "rtwdemo_capi/Gain1",
    "top_sig1", 0, 0, 0, 0, 0 },

  { 1, 0, "rtwdemo_capi/lu2d",
    "sig2_eg", 0, 0, 1, 0, 0 },

  {
    0, 0, (NULL), (NULL), 0, 0, 0, 0, 0
  }
};
```

Note To better understand the code, read the comments in the file. For example, notice the comment that begins on the third line in the preceding code. This comment lists the members of the `rtwCAPI_Signals` structure, in order. This tells you the order in which the assigned values for each member appear for a signal. In this example, the comment tells you that `signalName` is the fourth member of the structure. The following lines describe the first signal:

```
{ 0, 0, "rtwdemo_capi/Gain1",
  "top_sig1", 0, 0, 0, 0, 0 },
```

From these lines you infer that the name of the first signal is `top_sig1`.

Each array element, except the last, describes one output port for a block signal. The final array element is a sentinel, with all elements set to null values. For example, examine the second signal, described by the following code:

```
  { 1, 0, "rtwdemo_capi/lu2d",
    "sig2_eg", 0, 0, 1, 0, 0 },
```

This signal, named `sig2_eg`, is the output signal of the first port of the block `rtwdemo_capi/lu2d`. (This port is the first port because the zero-based index for `portNumber` displayed on the second line is assigned the value `0`.)

The address of this signal is given by `addrMapIndex`, which, in this example, is displayed on the first line as `1`. This provides an index into the `rtDataAddrMap` array, found later in `rtwdemo_capi_capi.c`:

```
/* Declare Data Addresses statically */
static void* rtDataAddrMap[] = {
  &rtwdemo_capi_B.top_sig1,            /* 0: Signal */
  &sig2_eg[0],                         /* 1: Signal */
  &rtwdemo_capi_DWork.top_state,       /* 2: Discrete State */
  &rtP_Ki,                             /* 3: Model Parameter */
  &rtP_Kp,                             /* 4: Model Parameter */
  &rtP_p1[0],                          /* 5: Model Parameter */
  &rtP_p2[0],                          /* 6: Model Parameter */
  &rtP_p3[0],                          /* 7: Model Parameter */
};
```

The index of `1` points to the second element in the `rtDataAddrMap` array. From the `rtDataAddrMap` array, you can infer that the address of this signal is `&sig2_eg[0]`.

This level of indirection supports multiple code instances of the same model. For multiple instances, the signal information remains constant, except for the address. In this case, the model is a single instance. Therefore, the `rtDataAddrMap` is declared statically. If you choose to generate reusable code, an initialize function is generated that initializes the addresses dynamically per instance. For details on generating reusable code, see "Entry-Point Functions and Scheduling" (Simulink Coder) and see "Configure Code Reuse Support" on page 30-15.

The `dataTypeIndex` provides an index into the `rtDataTypeMap` array, found later in `rtwdemo_capi_capi.c`, indicating the data type of the signal:

```
/* Data Type Map - use dataTypeMapIndex to access this structure */
static const rtwCAPI_DataTypeMap rtDataTypeMap[] = {
  /* cName, mwName, numElements, elemMapIndex, dataSize, slDataId, *
   * isComplex, isPointer */
  { "double", "real_T", 0, 0, sizeof(real_T), SS_DOUBLE, 0, 0 }
};
```

Because the index is `0` for `sig2_eg`, the index points to the first structure element in the array. You can infer that the data type of the signal is `double`. The value of `isComplex` is `0`, indicating that the signal is not complex. Rather than providing the data type information directly in the `rtwCAPI_Signals` structure, a level of indirection is introduced. The indirection allows multiple signals that share the same data type to point to one map structure, saving memory for each signal.

The `dimIndex` (dimensions index) provides an index into the `rtDimensionMap` array, found later in `rtwdemo_capi_capi.c`, indicating the dimensions of the signal. Because this index is `1` for `sig2_eg`, the index points to the second element in the `rtDimensionMap` array:

```
/* Dimension Map - use dimensionMapIndex to access elements of ths structure*/
static const rtwCAPI_DimensionMap rtDimensionMap[] = {
  /* dataOrientation, dimArrayIndex, numDims, vardimsIndex */
  { rtwCAPI_SCALAR, 0, 2, 0 },

  { rtwCAPI_VECTOR, 2, 2, 0 },
...
};
```

From this structure, you can infer that this is a nonscalar signal having a dimension of `2`. The `dimArrayIndex` value, 2, provides an index into `rtDimensionArray`, found later in `rtwdemo_capi_capi.c`:

```
/* Dimension Array- use dimArrayIndex to access elements of this array */
static const uint_T rtDimensionArray[] = {
  1,                                   /* 0 */
  1,                                   /* 1 */
  2,                                   /* 2 */
...
};
```

The `fxpIndex` (fixed-point index) provides an index into the `rtFixPtMap` array, found later in `rtwdemo_capi_capi.c`, indicating fixed-point information about the signal. Your code can use the scaling information to compute the real-world value of the signal, using the equation `V=SQ+B`, where V is "real-world" (that is, base-10) value, S is user-specified slope, Q is "quantized fixed-point value" or "stored integer," and B is user-specified bias. For details, see "Scaling" (Fixed-Point Designer).

Because this index is `0` for `sig2_eg`, the signal does not have fixed-point information. A fixed-point map index of zero means that the signal does not have fixed-point information.

The `sTimeIndex` (sample-time index) provides the index to the `rtSampleTimeMap` array, found later in `rtwdemo_capi_capi.c`, indicating task information about the

signal. If you log multirate signals or conditionally executed signals, the sampling information can be useful.

Note: *model*_capi.c (or .cpp) includes rtw_capi.h. A source file that references the rtBlockSignals array also must include rtw_capi.h.

C API States

The rtwCAPI_States structure captures state information including the state name, address, block path, type (continuous or discrete), data type information, dimensions information, fixed-point information, and sample-time information.

Here is the section of code in rtwdemo_capi_capi.c that provides information on C API states for the top model in rtwdemo_capi:

```
/* Block states information */
static const rtwCAPI_States rtBlockStates[] = {
  /* addrMapIndex, contStateStartIndex, blockPath,
   * stateName, pathAlias, dWorkIndex, dataTypeIndex, dimIndex,
   * fixPtIdx, sTimeIndex, isContinuous
   */
  { 2, -1, "rtwdemo_capi/Delay1",
    "top_state", "", 0, 0, 0, 0, 0, 0 },

  {
    0, -1, (NULL), (NULL), (NULL), 0, 0, 0, 0, 0, 0
  }
};
```

Each array element, except the last, describes a state in the model. The final array element is a sentinel, with all elements set to null values. In this example, the C API code for the top model displays one state:

```
{ 2, -1, "rtwdemo_capi/Delay1",
  "top_state", "", 0, 0, 0, 0, 0, 0 },
```

This state, named top_state, is defined for the block rtwdemo_capi/Delay1. The value of isContinuous is zero, indicating that the state is discrete rather than continuous. The other fields correspond to the like-named signal equivalents described in "C API Signals" on page 43-11, as follows:

- The address of the signal is given by addrMapIndex, which, in this example, is 2. This is an index into the rtDataAddrMap array, found later in

`rtwdemo_capi_capi.c`. Because the index is zero based, `2` corresponds to the third element in `rtDataAddrMap`, which is `&rtwdemo_capi_DWork.top_state`.

- The `dataTypeIndex` provides an index into the `rtDataTypeMap` array, found later in `rtwdemo_capi_capi.c`, indicating the data type of the parameter. The value 0 corresponds to a double, noncomplex parameter.
- The `dimIndex` (dimensions index) provides an index into the `rtDimensionMap` array, found later in `rtwdemo_capi_capi.c`. The value 0 corresponds to the first entry, which is `{ rtwCAPI_SCALAR, 0, 2, 0 }`.
- The `fixPtIndex` (fixed-point index) provides an index into the `rtFixPtMap` array, found later in `rtwdemo_capi_capi.c`, indicating fixed-point information about the parameter. As with the corresponding signal attribute, a fixed-point map index of zero means that the parameter does not have fixed-point information.

C API Root-Level Inputs and Outputs

The `rtwCAPI_Signals` structure captures root-level input/output information including the input/output name, address, block path, port number, data type information, dimensions information, fixed-point information, and sample-time information. (This structure also is used for block output signals, as previously described in "C API Signals" on page 43-11.)

Here is the section of code in `rtwdemo_capi_capi.c` that provides information on C API root-level inputs/outputs for the top model in `rtwdemo_capi`:

```
/* Root Inputs information */
static const rtwCAPI_Signals rtRootInputs[] = {
  /* addrMapIndex, sysNum, blockPath,
   * signalName, portNumber, dataTypeIndex, dimIndex, fxpIndex, sTimeIndex
   */
  { 3, 0, "rtwdemo_capi/In1",
    "", 1, 0, 0, 0, 0 },

  { 4, 0, "rtwdemo_capi/In2",
    "", 2, 0, 0, 0, 0 },

  { 5, 0, "rtwdemo_capi/In3",
    "", 3, 0, 0, 0, 0 },

  { 6, 0, "rtwdemo_capi/In4",
    "", 4, 0, 0, 0, 0 },

  {
    0, 0, (NULL), (NULL), 0, 0, 0, 0, 0
  }
};
```

```
/* Root Outputs information */
static const rtwCAPI_Signals rtRootOutputs[] = {
  /* addrMapIndex, sysNum, blockPath,
   * signalName, portNumber, dataTypeIndex, dimIndex, fxpIndex, sTimeIndex
   */
  { 7, 0, "rtwdemo_capi/Out1",
    "", 1, 0, 0, 0, 0 },

  { 8, 0, "rtwdemo_capi/Out2",
    "", 2, 0, 0, 0, 0 },

  { 9, 0, "rtwdemo_capi/Out3",
    "", 3, 0, 0, 0, 0 },

  { 10, 0, "rtwdemo_capi/Out4",
    "", 4, 0, 0, 0, 0 },

  { 11, 0, "rtwdemo_capi/Out5",
    "sig2_eg", 5, 0, 1, 0, 0 },

  { 12, 0, "rtwdemo_capi/Out6",
    "", 6, 0, 1, 0, 0 },

  {
    0, 0, (NULL), (NULL), 0, 0, 0, 0, 0
  }
};
```

For information about interpreting the values in the `rtwCAPI_Signals` structure, see the previous section "C API Signals" on page 43-11.

C API Parameters

The `rtwCAPI_BlockParameters` and `rtwCAPI_ModelParameters` structures capture parameter information including the parameter name, block path (for block parameters), address, data type information, dimensions information, and fixed-point information.

The `rtModelParameters` array contains entries for workspace variables that are referenced as tunable Simulink block parameters or Stateflow data of machine scope. For example, tunable parameters include `Simulink.Parameter` objects that use a storage class other than `Auto`. The Simulink Coder software assigns its elements only `NULL` or zero values in the absence of such data.

The setting that you choose for **Configuration Parameters** > **Optimization** > **Signals and Parameters** > **Default parameter behavior** determines how information is generated into the `rtBlockParameters` array in *`model`*`_capi.c` (or `.cpp`).

- If you set **Default parameter behavior** to `Tunable`, the `rtBlockParameters` array contains an entry for every modifiable parameter of every block in the model.

However, if you use a MATLAB variable or a tunable parameter to specify a block parameter, the block parameter does not appear in `rtBlockParameters`. Instead, the variable or tunable parameter appears in `rtModelParameters`.

- If you set **Default parameter behavior** to `Inlined`, the `rtBlockParameters` array is empty. The Simulink Coder software assigns its elements only `NULL` or zero values.

The last member of each array is a sentinel, with all elements set to null values.

Here is the `rtBlockParameters` array that is generated by default in `rtwdemo_capi_capi.c`:

```
/* Individual block tuning is not valid when inline parameters is *
 * selected. An empty map is produced to provide a consistent     *
 * interface independent  of inlining parameters.                 *
 */
static const rtwCAPI_BlockParameters rtBlockParameters[] = {
  /* addrMapIndex, blockPath,
   * paramName, dataTypeIndex, dimIndex, fixPtIdx
   */
  {
    0, (NULL), (NULL), 0, 0, 0
  }
};
```

In this example, only the final, sentinel array element is generated, with all members of the structure `rtwCAPI_BlockParameters` set to `NULL` and zero values. This is because **Default parameter behavior** is set to `Inlined` by default for the `rtwdemo_capi` example model. If you set **Default parameter behavior** to `Tunable`, the block parameters are generated in the `rtwCAPI_BlockParameters` structure. However, MATLAB variables and tunable parameters appear in the `rtwCAPI_ModelParameters` structure.

Here is the `rtModelParameters` array that is generated by default in `rtwdemo_capi_capi.c`:

```
/* Tunable variable parameters */
static const rtwCAPI_ModelParameters rtModelParameters[] = {
  /* addrMapIndex, varName, dataTypeIndex, dimIndex, fixPtIndex */
  { 2, TARGET_STRING("Ki"), 0, 0, 0 },

  { 3, TARGET_STRING("Kp"), 0, 0, 0 },

  { 4, TARGET_STRING("p1"), 0, 2, 0 },

  { 5, TARGET_STRING("p2"), 0, 3, 0 },
```

```
  { 6, TARGET_STRING("p3"), 0, 4, 0 },

  { 0, (NULL), 0, 0, 0 }
};
```

In this example, the `rtModelParameters` array contains entries for each variable that is referenced as a tunable Simulink block parameter.

For example, the `varName` (variable name) of the fourth parameter is `p2`. The other fields correspond to the like-named signal equivalents described in "C API Signals" on page 43-11, as follows:

- The address of the fourth parameter is given by `addrMapIndex`, which, in this example, is `5`. This is an index into the `rtDataAddrMap` array, found later in `rtwdemo_capi_capi.c`. Because the index is zero based, `5` corresponds to the sixth element in `rtDataAddrMap`, which is `rtP_p2`.
- The `dataTypeIndex` provides an index into the `rtDataTypeMap` array, found later in `rtwdemo_capi_capi.c`, indicating the data type of the parameter. The value 0 corresponds to a double, noncomplex parameter.
- The `dimIndex` (dimensions index) provides an index into the `rtDimensionMap` array, found later in `rtwdemo_capi_capi.c`. The value 3 corresponds to the fourth entry, which is `{ rtwCAPI_MATRIX_COL_MAJOR, 6, 2, 0 }`.
- The `fixPtIndex` (fixed-point index) provides an index into the `rtFixPtMap` array, found later in `rtwdemo_capi_capi.c`, indicating fixed-point information about the parameter. As with the corresponding signal attribute, a fixed-point map index of zero means that the parameter does not have fixed-point information.

For more information about tunable parameter storage in the generated code, see "Block Parameter Representation in the Generated Code" (Simulink Coder).

Map C API Data Structures to rtModel

The real-time model data structure encapsulates model data and associated information that describes the model fully. When you select the C API feature and generate code, the Simulink Coder code generator adds another member to the real-time model data structure generated in *`model`*`.h`:

```
/*
 * DataMapInfo:
 * The following substructure contains information regarding
 * structures generated in the model's C API.
```

```
 */
struct {
  rtwCAPI_ModelMappingInfo mmi;
} DataMapInfo;
```

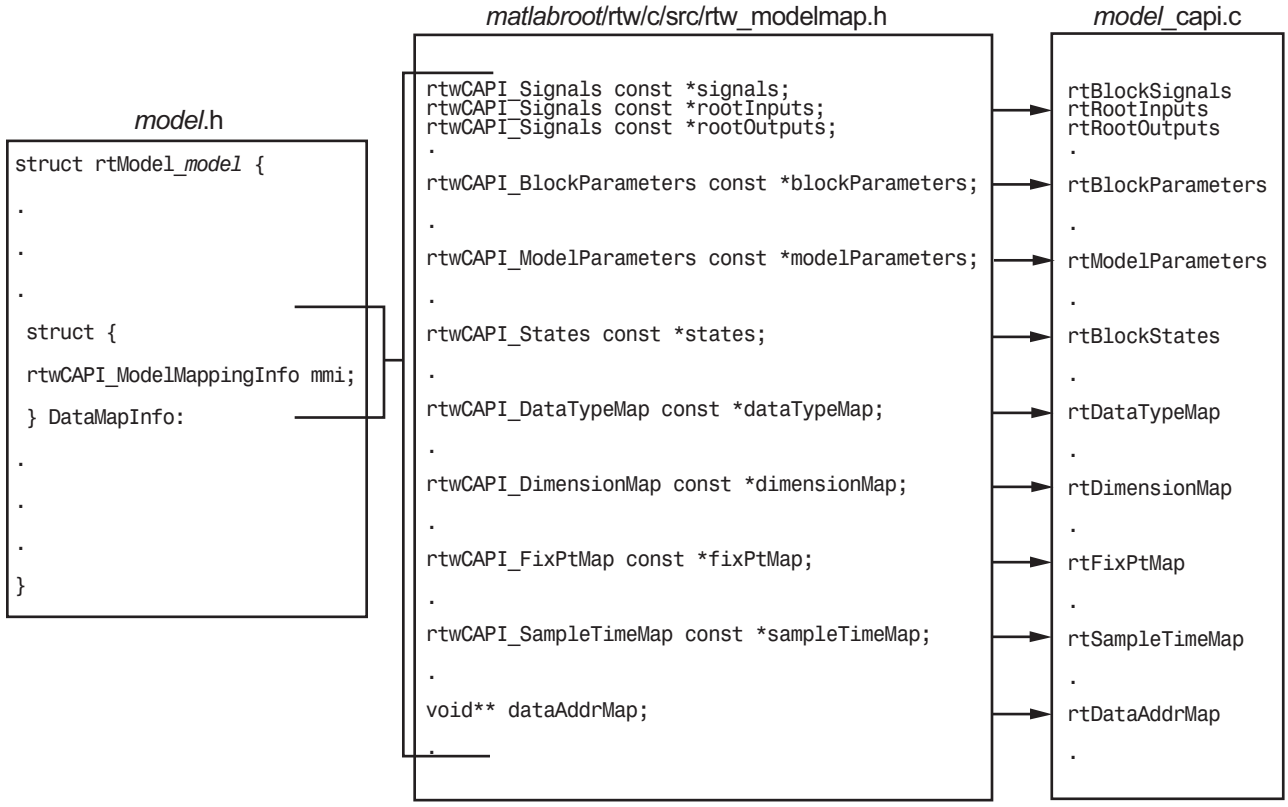
This member defines `mmi` (for model mapping information) of type `struct rtwCAPI_ModelMappingInfo`. The structure is located in *`matlabroot`*`/rtw/c/src/rtw_modelmap.h`. The `mmi` substructure defines the interface between the model and the C API files. More specifically, members of `mmi` map the real-time model data structure to the structures in *`model`*`_capi.c` (or `.cpp`).

Initializing values of `mmi` members to the arrays accomplishes the mapping, as shown in Map Model to C API Arrays of Structures. Each member points to one of the arrays of structures in the generated C API file. For example, the address of the `rtBlockSignals` array of structures is allocated to the first member of the `mmi` substructure in *`model`*`.c` (or `.cpp`), using the following code in the `rtw_modelmap.h` file:

```
/* signals */
struct {
    rtwCAPI_Signals const *signals;     /* Signals Array */
    uint_T                numSignals;   /* Num Signals   */
    rtwCAPI_Signals const *rootInputs;  /* Root Inputs array */
    uint_T               numRootInputs; /* Num Root Inputs  */
    rtwCAPI_Signals const *rootOutputs; /* Root Outputs array */
    uint_T               numRootOutputs;/* Num Root Outputs  */
} Signals;
```

The model initialize function in *`model`*`.c` (or `.cpp`) performs the initializing by calling the C API initialize function. For example, the following code is generated in the model initialize function for example model `rtwdemo_capi`:

```
/* Initialize DataMapInfo substructure containing ModelMap for C API */
rtwdemo_capi_InitializeDataMapInfo(rtwdemo_capi_M);
```



Map Model to C API Arrays of Structures

Note: This figure lists the arrays in the order that their structures appear in `rtw_modelmap.h`, which differs slightly from their generated order in *`model`*`_capi.c`.

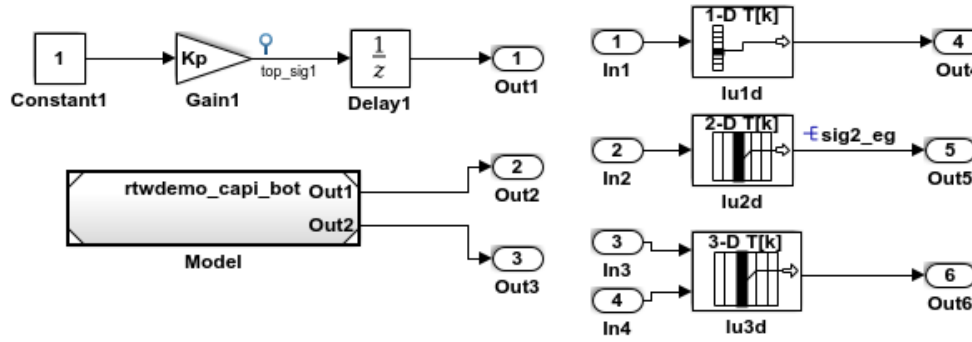
Generate C API Data Definition File for Exchanging Data with a Target System

This model illustrates the target-based C API for interfacing signals, parameters, and states in the generated code.

Open Example Model

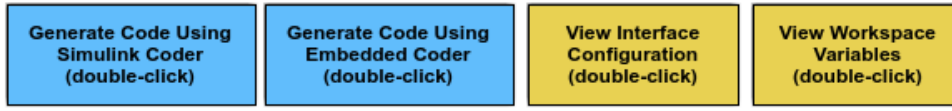
Open the example model `rtwdemo_capi`.

```
open_system('rtwdemo_capi');
```



This model illustrates the target-based C API for interfacing signals, parameters and states in the generated code. The C API is useful for real-time interaction with the data, without having to stop execution or recompile the generated code. Typically, a client/server protocol is set up from a host to a target using serial, TCP/IP, or dual-port memory connection. The purpose of this example is not the client/server protocol. Rather, this model shows the necessary data interface required by the C client/server programs.

You can enable the C API by selecting one or more C API options on the "Code Generation > Interface" pane of the Configuration Parameters dialog box. Any signal or parameter or state with an addressable storage class is placed in the C API data structure in <model>_capi.c. Note that signals, states and parameters in the referenced model can be accessed using C-API. So make sure that C-API is enabled for the referenced model.



The C API is useful for real-time interaction with application data, without having to stop execution or recompile the generated code. Typically, a client/server protocol is set up from a host to a target using serial, TCP/IP, or dual-port memory connection. The

purpose of this example is not the client/server protocol. Rather, this model shows the necessary data interface required by the C client/server programs.

You enable the C API by selecting one or more C API options on the **Code Generation > Interface** pane of the Configuration Parameters dialog box. Any signal or parameter or state with an addressable storage class is placed in the C API data structure in *`model`*`_capi.c`. Note that signals, states, and parameters in the referenced model can be accessed using C API. So make sure that C API is enabled for the referenced model.

C API Limitations

The C API feature has the following limitations.

- The C API does not support the following values for the `CodeFormat` TLC variable:
  - `S-Function`
  - `Accelerator_S-Function` (for accelerated simulation)
- For ERT-based targets, the C API requires that support for floating-point code be enabled.
- Local block output signals are not supported.
- Local Stateflow parameters are not supported.
- The following custom storage class objects are not supported:
  - Objects without the package `csc_registration` file
  - Grouped custom storage classes
  - Objects defined by using macros
  - `BitField` objects
  - `FileScope` objects
- Customized data placement is disabled when you are using the C API. The interface looks for global data declaration in *`model`*`.h` and *`model`*`_private.h`. Declarations placed in any other file by customized data placement result in code that does not compile.

Note Custom Storage Class objects work in code generation, only if you use the ERT target and clear the **Ignore custom storage classes** check box in the Configuration Parameters dialog box.

Related Examples

- "Access Signal, State, and Parameter Data During Execution" (Simulink Coder)

Use C API to Access Model Signals and States

This example helps you get started writing application code to interact with model signals and states. To get started writing application code to interact with model parameters, see "Use C API to Access Model Parameters" on page 43-30.

The C API provides you with the flexibility of writing your own application code to interact with model signals, states, root-level inputs/outputs, and parameters. Your target-based application code is compiled with the Simulink Coder generated code into an executable. The target-based application code accesses the C API structure arrays in *model*_capi.c (or .cpp). You might have host-based code that interacts with your target-based application code. Or, you might have other target-based code that interacts with your target-based application code. The files rtw_modelmap.h and rtw_capi.h, located in *matlabroot*/rtw/c/src (open), provide macros for accessing the structures in these arrays and their members.

Here is an example application that logs global signals and states in a model to a text file. This code is intended as a starting point for accessing signal and state addresses. You can extend the code to perform signal logging and monitoring, state logging and monitoring, or both.

This example uses the following macro and function interfaces:

- rtmGetDataMapInfo macro

  Accesses the model mapping information (MMI) substructure of the real-time model structure. In the following macro call, rtM is the pointer to the real-time model structure in *model*.c (or .cpp):

  ```
  rtwCAPI_ModelMappingInfo* mmi = &(rtmGetDataMapInfo(rtM).mmi);
  ```

- rtmGetTPtr macro

  Accesses the absolute time information for the base rate from the timing substructure of the real-time model structure. In the following macro call, rtM is the pointer to the real-time model structure in *model*.c (or .cpp):

  ```
  rtmGetTPtr(rtM)
  ```

- Custom functions capi_StartLogging, capi_UpdateLogging, and capi_TerminateLogging, provided via the files rtwdemo_capi_datalog.h and rtwdemo_capi_datalog.c. These files are located in *matlabroot*/toolbox/rtw/rtwdemos (open).

- `capi_StartLogging` initializes signal and state logging.
- `capi_UpdateLogging` logs a signal and state value at each time step.
- `capi_TerminateLogging` terminates signal and state logging and writes the logged values to a text file.

You can integrate these custom functions into generated model code using one or more of the following methods:

- **Code Generation** > **Custom Code** pane of the Configuration Parameters dialog box
- Custom Code library blocks
- TLC custom code functions

This tutorial uses the **Code Generation** > **Custom Code** pane and the System Outputs block from the Custom Code library to insert calls to the custom functions into *`model`*`.c` (or `.cpp`), as follows:

- `capi_StartLogging` is called in the *`model`*`_initialize` function.
- `capi_UpdateLogging` is called in the *`model`*`_step` function.
- `capi_TerminateLogging` is called in the *`model`*`_terminate` function.

The following excerpts of generated code from *`model`*`.c` (rearranged to reflect their order of execution) show how the function interfaces are used.

```
void rtwdemo_capi_initialize(void)
{
...
  /* user code (Initialize function Body) */

  /* C API Custom Logging Function: Start Signal and State logging via C API.
   * capi_StartLogging: Function prototype in rtwdemo_capi_datalog.h
   */
  {
    rtwCAPI_ModelMappingInfo *MMI = &(rtmGetDataMapInfo(rtwdemo_capi_M).mmi);
    printf("** Started state/signal logging via C API **\n");
    capi_StartLogging(MMI, MAX_DATA_POINTS);
  }
...
}
...
/* Model step function */
void rtwdemo_capi_step(void)
{
...
  /* user code (Output function Trailer) */

  /* System '<Root>' */
```

```
  /* C API Custom Logging Function: Update Signal and State logging buffers.
   * capi_UpdateLogging: Function prototype in rtwdemo_capi_datalog.h
   */
  {
    rtwCAPI_ModelMappingInfo *MMI = &(rtmGetDataMapInfo(rtwdemo_capi_M).mmi);
    capi_UpdateLogging(MMI, rtmGetTPtr(rtwdemo_capi_M));
  }
...
}
...
/* Model terminate function */
void rtwdemo_capi_terminate(void)
{
  /* user code (Terminate function Body) */

  /* C API Custom Logging Function: Dump Signal and State buffers into a text file.
   * capi_TerminateLogging: Function prototype in rtwdemo_capi_datalog.h
   */
  {
    capi_TerminateLogging("rtwdemo_capi_ModelLog.txt");
    printf("** Finished state/signal logging. Created rtwdemo_capi_ModelLog.txt **\n");
  }
}
```

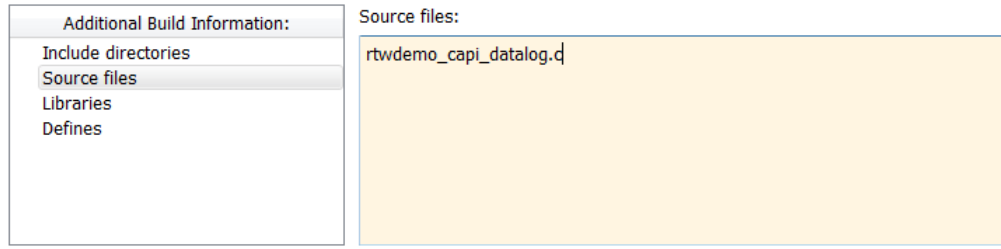
The following procedure illustrates how you can use the C API macro and function interfaces to log global signals and states in a model to a text file.

1. At the MATLAB command line, enter `rtwdemo_capi` to open the example model.
2. Save the top model `rtwdemo_capi` and the referenced model `rtwdemo_capi_bot` to the same writable work folder.
3. Open the Configuration Parameters dialog box.
4. If you are licensed for Embedded Coder software and you want to use the `ert.tlc` target instead of the default `grt.tlc`, go to the **Code Generation** pane and use the **System target file** field to select an `ert.tlc` target. Make sure that you also select `ert.tlc` for the referenced model `rtwdemo_capi_bot`.
5. In the top model, go to the **Code Generation** > **Interface** pane.
   a. In the **Data exchange interface** subgroup, verify that the model options **Generate C API for: signals** and **Generate C API for: states** are selected. This example also leaves **Generate C API for: parameters** selected.
   b. If you are using the `ert.tlc` system target file, verify that the option **Support: complex numbers** is selected.
   c. Click the **All Parameters** tab, and select the **MAT-file logging** option.
   d. Click **Apply**.
   e. Update configuration parameter settings in the referenced model, `rtwdemo_capi_bot`, to match changes you made in the top model.

6 Use the **Custom Code** pane to embed your custom application code in the generated code. Select the **Custom Code** pane, and then click **Include directories**. The **Include directories** input field is displayed.

7 In the **Include directories** field, type *`matlabroot`*`/toolbox/rtw/rtwdemos`, where *`matlabroot`* represents the root of your MATLAB installation folder. (If you are specifying a Windows path that contains a space, place the text inside double quotes.)

8 In the **Additional Build Information** subpane, click **Source files**, and type `rtwdemo_capi_datalog.c`.



9 In the **Include custom C code in generated** subpane, click **Source file**, and type or copy and paste the following include statement:

```
#include "rtwdemo_capi_datalog.h"
```

10 In the **Initialize function** field, type or copy and paste the following application code:

```
/* C API Custom Logging Function: Start Signal and State logging via C API.
 * capi_StartLogging: Function prototype in rtwdemo_capi_datalog.h
 */
{
  rtwCAPI_ModelMappingInfo *MMI = &(rtmGetDataMapInfo(rtwdemo_capi_M).mmi);
  printf("** Started state/signal logging via C API **\n");
  capi_StartLogging(MMI, MAX_DATA_POINTS);
}
```

Note: If you renamed the top model `rtwdemo_capi`, update the name `rtwdemo_capi_M` in the application code to reflect the new model name.

---

11 In the **Terminate function** field, type or copy and paste the following application code:

```
/* C API Custom Logging Function: Dump Signal and State buffers into a text file.
 * capi_TerminateLogging: Function prototype in rtwdemo_capi_datalog.h
 */
```

```
  {
capi_TerminateLogging("rtwdemo_capi_ModelLog.txt");
printf("** Finished state/signal logging. Created rtwdemo_capi_ModelLog.txt **\n");
  }
```

Click **Apply**.

12 In the MATLAB Command Window, enter `custcode` to open the Simulink Coder Custom Code library. At the top level of the `rtwdemo_capi` model, add a System Outputs block.

13 Double-click the System Outputs block to open the System Outputs Function Custom Code dialog box. In the **System Outputs Function Exit Code** field, type or copy and paste the following application code:

```
/* C API Custom Logging Function: Update Signal and State logging buffers.
 * capi_UpdateLogging: Function prototype in rtwdemo_capi_datalog.h
 */
{
  rtwCAPI_ModelMappingInfo *MMI = &(rtmGetDataMapInfo(rtwdemo_capi_M).mmi);
  capi_UpdateLogging(MMI, rtmGetTPtr(rtwdemo_capi_M));
}
```

Note: If you renamed the top model `rtwdemo_capi`, update two instances of the name `rtwdemo_capi_M` in the application code to reflect the new model name.

Click **OK**.

14 On the **Code Generation** pane, verify that the **Generate code only** check box is *cleared*.

Press **Ctrl+B** to build the model and generate an executable file. For example, on a Windows system, the build generates the executable file `rtwdemo_capi.exe` in your current working folder.

15 In the MATLAB Command Window, enter the command `!rtwdemo_capi` to run the executable file. During execution, signals and states are logged using the C API and then written to the text file `rtwdemo_capi_ModelLog.txt` in your current working folder.

```
>> !rtwdemo_capi

** starting the model **
** Started state/signal logging via C API **
** Logging 2 signal(s) and 1 state(s). In this demo, only scalar named
 signals/states are logged **
** Finished state/signal logging. Created rtwdemo_capi_ModelLog.txt **
```

16 Examine the text file in the MATLAB editor or other text editor. Here is an excerpt of the signal and state logging output.

```
******** Signal Log File ********

Number of Signals Logged: 2
Number of points (time steps) logged: 51

Time           bot_sig1 (Referenced Model)            top_sig1
0              70                                     4
0.2            70                                     4
0.4            70                                     4
0.6            70                                     4
0.8            70                                     4
1              70                                     4
1.2            70                                     4
1.4            70                                     4
1.6            70                                     4
1.8            70                                     4
2              70                                     4
...

******** State Log File ********

Number of States Logged: 1
Number of points (time steps) logged: 51

Time           bot_state (Referenced Model)
0              0
0.2            70
0.4            35
0.6            52.5
0.8            43.75
1              48.13
1.2            45.94
1.4            47.03
1.6            46.48
1.8            46.76
2              46.62
...
```

Use C API to Access Model Parameters

This example helps you get started writing application code to interact with model parameters. To get started writing application code to interact with model signals and states, see "Use C API to Access Model Signals and States" on page 43-24.

The C API provides you with the flexibility of writing your own application code to interact with model signals, states, root-level inputs/outputs, and parameters. Your target-based application code is compiled with the Simulink Coder generated code into an executable. The target-based application code accesses the C API structure arrays in *`model`*`_capi.c` (or `.cpp`). You might have host-based code that interacts with your target-based application code. Or, you might have other target-based code that interacts with your target-based application code. The files `rtw_modelmap.h` and `rtw_capi.h`, located in *`matlabroot`*`/rtw/c/src` (open), provide macros for accessing the structures in these arrays and their members.

Here is an example application that prints the parameter values of tunable parameters in a model to the standard output. This code is intended as a starting point for accessing parameter addresses. You can extend the code to perform parameter tuning. The application:

- Uses the `rtmGetDataMapInfo` macro to access the mapping information in the `mmi` substructure of the real-time model structure

  ```
  rtwCAPI_ModelMappingInfo* mmi = &(rtmGetDataMapInfo(rtM).mmi);
  ```

  where `rtM` is the pointer to the real-time model structure in *`model`*`.c` (or `.cpp`).
- Uses `rtwCAPI_GetNumModelParameters` to get the number of model parameters in mapped C API:

  ```
  uint_T nModelParams = rtwCAPI_GetNumModelParameters(mmi);
  ```
- Uses `rtwCAPI_GetModelParameters` to access the array of model parameter structures mapped in C API:

  ```
  rtwCAPI_ModelParameters* capiModelParams = \
   rtwCAPI_GetModelParameters(mmi);
  ```
- Loops over the `capiModelParams` array to access individual parameter structures. A call to the function `capi_PrintModelParameter` displays the value of the parameter.

The example application code is provided below:

```
{
/* Get CAPI Mapping structure from Real-Time Model structure */
rtwCAPI_ModelMappingInfo* capiMap = \
&(rtmGetDataMapInfo(rtwdemo_capi_M).mmi);

/* Get number of Model Parameters from capiMap */
uint_T nModelParams = rtwCAPI_GetNumModelParameters(capiMap);
printf("Number of Model Parameters: %d\n", nModelParams);

/* If the model has Model Parameters, print them using the
application capi_PrintModelParameter */
if (nModelParams == 0) {
   printf("No Tunable Model Parameters in the model \n");
}
else {
   unsigned int idx;

   for (idx=0; idx < nModelParams; idx++) {
      /* call print utility function */
      capi_PrintModelParameter(capiMap, idx);
   }
}
}
```

The print utility function is located in *`matlabroot`*`/rtw/c/src/rtw_capi_examples.c`. This file contains utility functions for accessing the C API structures.

To become familiar with the example code, try building a model that displays the tunable block parameters and MATLAB variables. You can use `rtwdemo_capi`, the C API example model. The following steps apply to both `grt.tlc` and `ert.tlc` targets, unless otherwise indicated.

1. At the MATLAB command line, enter `rtwdemo_capi` to open the example model.
2. Save the top model `rtwdemo_capi` and the referenced model `rtwdemo_capi_bot` to the same writable work folder.
3. If you are licensed for Embedded Coder software and you want to use the `ert.tlc` target instead of the default `grt.tlc`, go to the **Code Generation** pane of the Configuration Parameters dialog box and use the **System target file** field to select an `ert.tlc` target. Make sure that you also select `ert.tlc` for the referenced model `rtwdemo_capi_bot`.
4. Go to the **Code Generation** > **Interface** pane.

a In the **Data exchange interface** subgroup, verify that the model option **Generate C API for: parameters** is selected.

b If you are using the `ert.tlc` system target file, verify that the option **Support: complex numbers** is selected.

c Click the **All Parameters** tab, and select the **MAT-file logging** option.

d Click **Apply**.

e Update configuration parameter settings in the referenced model, `rtwdemo_capi_bot`, to match changes you made in the top model.

5 Use the **Custom Code** pane to embed your custom application code in the generated code. Select the **Custom Code** pane, and then click **Initialize function**. The **Initialize function** input field is displayed.

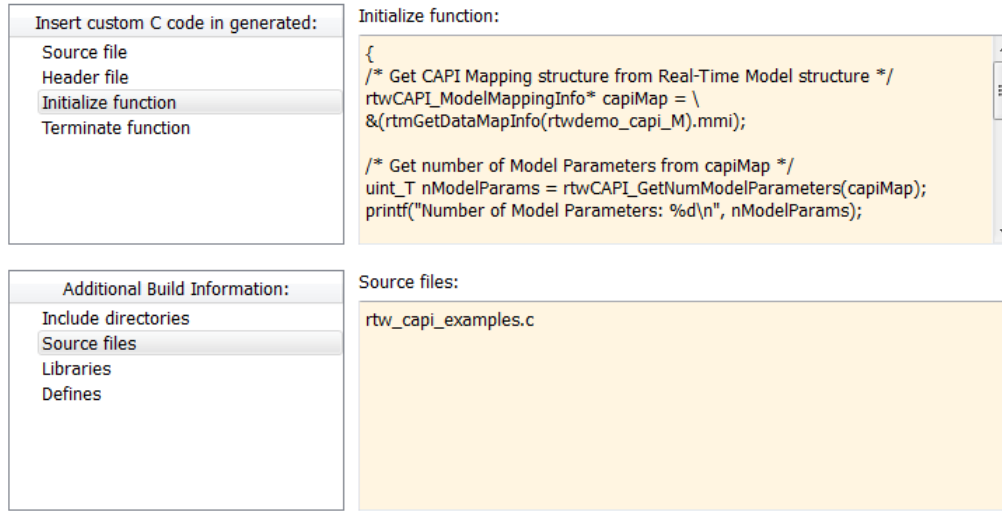
6 In the **Initialize function** input field, type or copy and paste the example application code shown above step 1. This embeds the application code in the *`model`*`_initialize` function.

Note: If you renamed the top model `rtwdemo_capi`, update the name `rtwdemo_capi_M` in the application code to reflect the new model name.

---

7 Click **Include directories**, and type *`matlabroot`*`/rtw/c/src`, where *`matlabroot`* represents the root of your MATLAB installation folder. (If you are specifying a Windows path that contains a space, place the text inside double quotes.)

8 In the **Additional Build Information** subpane, click **Source files**, and type `rtw_capi_examples.c`.



Click **Apply**.

9 On the **Code Generation** pane, verify that the **Generate code only** check box is *cleared*.

Press **Ctrl+B** to build the model and generate an executable file. For example, on a Windows system, the build generates the executable file `rtwdemo_capi.exe` in your current working folder.

10 In the MATLAB Command Window, enter `!rtwdemo_capi` to run the executable file. Running the program displays parameter information in the Command Window.

```
>> !rtwdemo_capi

** starting the model **
Number of Model Parameters: 5
Ki =
     7
Kp =
     4
p1 =
    0.15
    0.36
    0.81
```

```
p2 =
    0.09    0.75    0.57
    0.13    0.96    0.059
p3 =
ans(:,:,1) =
    0.23    0.82    0.04    0.64
    0.35    0.01    0.16    0.73

ans(:,:,2) =
    0.64    0.54    0.74    0.68
    0.45    0.29    0.18    0.18
```

44

ASAP2 Data Measurement and Calibration in Simulink Coder

Export ASAP2 File for Data Measurement and Calibration

The ASAM MCD-2 MC standard, also known as ASAP2, is a data definition standard proposed by the Association for Standardization of Automation and Measuring Systems (ASAM). ASAP2 is a non-object-oriented description of the data used for measurement, calibration, and diagnostic systems. For more information on ASAM and the ASAM MCD-2 MC (ASAP2) standard, see the ASAM Web site at `http://www.asam.net`.

The Simulink Coder product lets you export an ASAP2 file containing information about your model during the code generation process.

You can run an interactive example of ASAP2 file generation. To open the example at the MATLAB command prompt, enter the following command:

```
rtwdemo_asap2
```

Note: Simulink Coder support for ASAP2 file generation is version-neutral. By default, the software generates ASAP2 version 1.31 format, but the generated model information is generally compatible with all ASAP2 versions. ASAP2 file generation also is neutral with respect to the specific needs of ASAP2 measurement and calibration tools. The software provides customization APIs that you can use to customize ASAP2 file generation to generate any ASAP2 version and to meet the specific needs of your ASAP2 tools.

In this section...

What You Should Know

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

- ASAM and the ASAP2 standard and terminology. See the ASAM Web site at http://www.asam.net.
- Simulink data objects. Data objects are used to supply information not contained in the model. For an overview, see "Data Objects" (Simulink).
- Storage and representation of signals and parameters in generated code. See "Data Representation" (Simulink Coder).
- If you are licensed for Embedded Coder, see also the Embedded Coder topic "Data Representation".

Targets Supporting ASAP2

ASAP2 file generation is available to all Simulink Coder target configurations. You can select these target configurations from the System Target File Browser. For example,

- The `Generic Real-Time Target` (`grt.tlc`) lets you generate an ASAP2 file as part of the code generation and build process.
- The `Embedded Coder` (`ert.tlc`) target selections also lets you generate an ASAP2 file as part of the code generation and build process.
- The `ASAM-ASAP2 Data Definition Target` (`asap2.tlc`) lets you generate only an ASAP2 file, without building an executable.

Procedures for generating ASAP2 files by using these target configurations are given in "Generate an ASAP2 File" on page 44-9.

Define ASAP2 Information

Define ASAP2 Information for Parameters and Signals

The ASAP2 file generation process requires information about parameters and signals in your model. Some of this information is contained in the model itself. You must supply the rest by using Simulink data objects and corresponding properties.

You can use built-in Simulink data objects to provide the information. For example, you can use `Simulink.Signal` objects to provide MEASUREMENT information and `Simulink.Parameter` objects to provide CHARACTERISTIC information. Also, you can use data objects from data classes that are derived from `Simulink.Signal` and `Simulink.Parameter` to provide the information. For details, see "Data Objects" (Simulink).

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

Data Attribute	Defined In	Property Name
Name (symbol)	Data object	Inherited from the handle of the data object to which parameter or signal name resolves
Description	Data object	`Description`
Data type	Model	Not applicable
Scaling (if fixed-point data type)	Model	Data type (for signals) Inherited from value (for parameters)
Minimum allowable value	Data object	`Min`
Maximum allowable value	Data object	`Max`
Unit	Data object	`Unit`
Memory address (optional)	Data object	`MemoryAddress_ASAP2` (optional; see "Memory Address Attribute" on page 44-4.)

Memory Address Attribute

If the memory address attribute is unknown before code generation, the code generator inserts `ECU Address` placeholder text in the generated ASAP2 file. You can substitute an actual address for the placeholder by postprocessing the generated file. See the file *`matlabroot`*`/toolbox/rtw/targets/asap2/asap2/asap2post.m` for an example.

`asap2post.m` parses through the linker map file that you provide and replaces the `ECU Address` placeholders in the ASAP2 file with the actual memory addresses. Since linker map files vary from compiler to compiler, you might need to modify the regular expression code in `asap2post.m` to match the format of the linker map you use.

Note: If Embedded Coder is licensed and installed on your system, and if you are generating ELF (Executable and Linkable Format) files for your embedded target, you can use the `rtw.asap2SetAddress` function to automate ECU address replacement. For more information, see "Automatic ECU Address Replacement for ASAP2 Files (Embedded Coder)" on page 44-6.

If the memory address attribute is known before code generation, it can be defined in the data object. By default, the `MemoryAddress_ASAP2` property does not exist in the `Simulink.Signal` or `Simulink.Parameter` data object classes. If you want to add the attribute, add a property called `MemoryAddress_ASAP2` to a custom class that is a subclass of the `Simulink` or `ASAP2` class. For information on subclassing Simulink data classes, see "Define Data Classes" (Simulink).

Note In previous releases, for ASAP2 file generation, you had to define objects explicitly as `ASAP2.Signal` and `ASAP2.Parameter`. This is no longer a limitation. As explained above, you can use built-in Simulink objects for generating an ASAP2 file. If you have been using an earlier release, you can continue to use the ASAP2 objects. If one of these ASAP2 objects was created in the previous release, and you use it in this release, the MATLAB Command Window displays a warning the first time the objects are loaded.

The following table indicates the Simulink object properties that have replaced the ASAP2 object properties of the previous release:

Differences Between ASAP2 and Simulink Parameter and Signal Object Properties

ASAP2 Object Properties (Previous)	Simulink Object Properties (Current)
LONGIG_ASAP2	Description
PhysicalMin_ASAP2	Min
PhysicalMax_ASAP2	Max
Units_ASAP2	Unit

Automatic ECU Address Replacement for ASAP2 Files (Embedded Coder)

If Embedded Coder is licensed and installed on your system, and if you are generating ELF (Executable and Linkable Format) files for your embedded target, you can use the `rtw.asap2SetAddress` function to automate the replacement of `ECU Address` placeholder memory address values with actual addresses in the generated ASAP2 file.

If the memory address attribute is unknown before code generation, the code generator inserts `ECU Address` placeholder text in the generated ASAP2 file, as shown in the example below.

```
/begin CHARACTERISTIC
  /* Name            */ Ki
  /* Long Identifier */ ""
  /* Type            */ VALUE
  /* ECU Address     */ 0x0
                       /*ECU_Address@Ki@ */
```

To substitute actual addresses for the `ECU Address` placeholders, process the generated ASAP2 file using the `rtw.asap2SetAddress` function. The general syntax is as follows:

```
rtw.asap2SetAddress(ASAP2File, InfoFile)
```

where the arguments are character vectors specifying the name of the generated ASAP2 file and the name of the generated executable ELF file or DWARF debug information file for the model. When called, `rtw.asap2SetAddress` extracts the actual ECU address from the specified ELF or DWARF file and replaces the placeholder in the ASAP2 file with the actual address, for example:

```
/begin CHARACTERISTIC
  /* Name            */ Ki
  /* Long Identifier */ ""
  /* Type            */ VALUE
  /* ECU Address     */ 0x40009E60
```

Define ASAP2 Information for Lookup Tables

Simulink Coder software generates ASAP2 descriptions for lookup table data and its breakpoints. The software represents 1-D table data as `CURVE` information, 2-D table data as `MAP` information, and breakpoints as `AXIS_DESCR` and `AXIS_PTS` information. You can model lookup tables using one of the following Simulink Lookup Table blocks:

- Direct Lookup Table (n-D) — 1 and 2 dimensions
- Interpolation Using Prelookup — 1 and 2 dimensions
- 1–D Lookup Table
- 2–D Lookup Table
- n-D Lookup Table — 1 and 2 dimensions

The software supports the following types of lookup table breakpoints (axis points):

Breakpoint Type	Generates
Tunable and shared among multiple table axes (common axis)	`COM_AXIS`
Fixed and nontunable (fixed axis)	One of these variants of `FIX_AXIS`: • `FIX_AXIS_PAR` if breakpoints are integers with equidistant spacing and the equidistant spacing is a power of two • `FIX_AXIS_PAR_DIST` if breakpoints are integers with equidistant spacing • `FIX_AXIS_PAR_LIST` if breakpoints are integers with non-equidistant spacing
Tunable but not shared among multiple tables (standard axis)	`STD_AXIS`

When you configure the blocks for ASAP2 code generation:

- For table data, use a `Simulink.Parameter` data object with a non-`Auto` storage class.
- For tunable breakpoint data that is shared among multiple table axes (`COM_AXIS`), use a `Simulink.Parameter` data object with a non-`Auto` storage class.
- For fixed, nontunable breakpoint data (`FIX_AXIS`), use workspace variables or arrays specified in the block parameters dialog box. The breakpoints should be stored as integers in the code, so the data type should be a built-in integer type (`int8`, `int16`, `int32`, `uint8`, `uint16`, or `uint32`), a fixed-point data type, or an equivalent alias type.
- For tunable breakpoint data that is not shared among multiple tables (STD_AXIS):

  1. Create a `Simulink.Bus` object to define the `struct` packaging (names and order of the fields). The fields of the parameter structure must correspond to the lookup table data and each axis of the lookup table block. For example, in an n-D Lookup Table block with 2 dimensions, the structure must contain only three fields. This bus object describes the record layout for the lookup characteristic.
  2. Create a `Simulink.Parameter` object to represent a tunable parameter.
  3. Create table and axis values.
  4. Optionally, specify the **Units**, **Minimum**, and **Maximum** properties for the parameter object. The properties will be applied to table data only.

Here is an example of an n-D Lookup Table record generated into an ASAP2 file in Standard Axis format:

```
/begin CHARACTERISTIC
  /* Name */  STDAxisParam
  ...
  /* Record Layout */   Lookup1D_X_WORD_Y_FLOAT32_IEEE
  ...
  begin AXIS_DESCR
    /* Description of X-Axis Points */
    /* Axis Type */      STD_AXIS
    ...
  /end AXIS_DESCR
/end CHARACTERISTIC

/begin RECORD_LAYOUT Lookup1D_X_WORD_Y_FLOAT32_IEEE
  AXIS_PTS_X 1 WORD INDEX_INCR DIRECT
```

```
  FNC_VALUES 2 FLOAT32_IEEE COLUMN_DIR DIRECT
/end   RECORD_LAYOUT
```

Note: The example model `rtwdemo_asap2` illustrates ASAP2 file generation for Lookup Table blocks, including both tunable (COM_AXIS) and fixed (FIX_AXIS) lookup table breakpoints.

Generate an ASAP2 File

About Generating ASAP2 Files

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

- Use the Generic Real-Time Target or a Embedded Coder target to generate an ASAP2 file as part of the code generation and build process.
- Use the ASAM-ASAP2 Data Definition Target to generate only an ASAP2 file, without building an executable.

This section discusses how to generate an ASAP2 file by using the targets that have built-in ASAP2 support. For an example, see the ASAP2 example model `rtwdemo_asap2`.

Use GRT or ERT Target

The procedure for generating the ASAP2 data definition for a model using the Generic Real-Time Target or an Embedded Coder target is as follows:

1. Create the desired model. Use parameter names and signal labels to refer to corresponding CHARACTERISTIC records and MEASUREMENT records, respectively.
2. Create `Simulink.Parameter` and `Simulink.Signal` objects in the MATLAB workspace and reference them from parameters and signals in the model. A convenient way of creating multiple signal and parameter data objects is to use the

Data Object Wizard. Alternatively, you can create data objects one at a time from the MATLAB command line. For details on how to use the Data Object Wizard, see "Create Data Objects for a Model Using Data Object Wizard" (Simulink).

3 For each data object, configure the **Storage class** property to a setting other than `Auto`, `FileScope`, or `SimulinkGlobal`. This setting declares the data object as global in the generated code. For example, a storage class setting of `ExportedGlobal` configures the data object as unstructured global in the generated code.

Note: The data object is not represented in the ASAP2 file if any of the following conditions exist:

- You set the storage class to `Auto`, `FileScope`, or `SimulinkGlobal`.
- You set the storage class to `Custom` and custom storage class settings cause the code generator to generate a macro or non-addressable variable.

---

4 Configure the remaining properties as desired for each data object.

5 On the **Code Generation** pane, click **Browse** to open the System Target File Browser. In the browser, select `grt.tlc` or an ERT based target file and click **OK**.

6 On the **Code Generation** > **Interface** pane, in the **Data exchange interface** subgroup, select **ASAP2 interface**.

7 Select the **Generate code only** check box on the **Code Generation** pane.

8 Click **Apply**.

9 Press **Ctrl+B** to build the model.

The Simulink Coder code generator writes the ASAP2 file to the build folder. By default, the file is named *`model`*`.a2l`, where *`model`* is the name of the model. The ASAP2 setup file controls the ASAP2 file name. For details, see "Customize Generated ASAP2 File" on page 69-2.

Use the ASAM-ASAP2 Data Definition Target

The procedure for generating the ASAP2 data definition for a model using the ASAM-ASAP2 Data Definition Target is as follows:

1 Create the desired model. Use parameter names and signal labels to refer to corresponding CHARACTERISTIC records and MEASUREMENT records, respectively.

2. Create `Simulink.Parameter` and `Simulink.Signal` objects in the MATLAB workspace and reference them from parameters and signals in the model. A convenient way of creating multiple signal and parameter data objects is to use the Data Object Wizard. Alternatively, you can create data objects one at a time from the MATLAB command line. For details on how to use the Data Object Wizard, see "Create Data Objects for a Model Using Data Object Wizard" (Simulink).
3. For each data object, configure the **Storage class** property to a setting other than `Auto`, `FileScope`, or `SimulinkGlobal`. This setting declares the data object as global in the generated code. For example, a storage class setting of `ExportedGlobal` configures the data object as unstructured global in the generated code.

   Note: The data object is not represented in the ASAP2 file if any of the following conditions exist:

   - You set the storage class to `Auto`, `FileScope`, or `SimulinkGlobal`.
   - You set the storage class to `Custom` and custom storage class settings cause the code generator to generate a macro or non-addressable variable.

   ---

4. Configure the remaining properties as desired for each data object.
5. On the **Code Generation** pane, click **Browse** to open the System Target File Browser. In the browser, select `asap2.tlc` and click **OK**.
6. Select the **Generate code only** check box on the **Code Generation** pane.
7. Click **Apply**.
8. Press **Ctrl+B**.

   The Simulink Coder code generator writes the ASAP2 file to the build folder. By default, the file is named *`model`*`.a2l`, where *`model`* is the name of the model. The ASAP2 setup file controls the ASAP2 file name. For details, see "Customize Generated ASAP2 File" on page 69-2.

Generate ASAP2 Files for Referenced Models

The build process can generate an ASAP2 file for each referenced model in a model reference hierarchy. In the generated ASAP2 file, MEASUREMENT records represent signals and states inside the referenced model.

To generate ASAP2 files for referenced models, select ASAP2 file generation for the top model and for each referenced model in the reference hierarchy. For example, if you are

using the Generic Real-Time Target or an Embedded Coder target, follow the procedure described in "Use GRT or ERT Target" on page 44-9 for the top model and each referenced model.

Merge ASAP2 Files for Top and Referenced Models

Use function `rtw.asap2MergeMdlRefs` to merge the ASAP2 files generated for top and referenced models. The function has the following syntax:

```
[status,info]=rtw.asap2MergeMdlRefs(topModelName,asap2FileName)
```

- `topModelName` is the name of the model containing one or more referenced models.
- `asap2FileName` is the name you specify for the merged ASAP2 file.
- *Optional:*: `status` returns false (logical 0) if the merge completes and true (logical 1) otherwise.
- *Optional:*: `info` returns additional information about merge failure if `status` is true. Otherwise, it returns an empty character vector.

Consider the following example.

```
[status,info]=rtw.asap2MergeMdlRefs('myTopMdl','merged.a2l')
```

This command merges the ASAP2 files generated for the top model `myTopMdl` and its referenced models in the file `merged.a2l`.

The example model `rtwdemo_asap2` includes an example of merging ASAP2 files.

Structure of the ASAP2 File

The following table outlines the basic structure of the ASAP2 file and describes the Target Language Compiler (TLC) functions and files used to create each part of the file:

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

File Section	Contents of asap2main.tlc	TLC File Containing Function Definition
File header	`%<ASAP2UserFcnWriteFileHead()>`	`asap2userlib.tlc`
`/begin PROJECT ""`	**`/begin PROJECT`** `"%<ASAP2ProjectName>"`	`asap2setup.tlc`

File Section	Contents of asap2main.tlc	TLC File Containing Function Definition
/begin HEADER "" HEADER contents	**/begin HEADER**"%<ASAP2HeaderName>" %<ASAP2UserFcnWriteHeader()>	asap2setup.tlc asap2userlib.tlc
/end HEADER	**/end HEADER**	
/begin MODULE "" MODULE contents:	**/begin MODULE** "%<ASAP2ModuleName>"}	asap2setup.tlc asap2userlib.tlc
- A2ML - MOD_PAR - MOD_COMMON ...	%<ASAP2UserFcnWriteHardwareInterface()>	
Model-dependent MODULE contents:	%<SLibASAP2WriteDynamicContents()> Calls user-defined functions:	asap2lib.tlc
- RECORD_LAYOUT - CHARACTERISTIC - ParameterGroups - ModelParameters	...WriteRecordLayout_TemplateName() ...WriteCharacteristic_TemplateName() ...WriteCharacteristic_Scalar()	user/templates/...
- MEASUREMENT - ExternalInputs - BlockOutputs	...WriteMeasurement()	asap2userlib.tlc
- COMPU_METHOD	...WriteCompuMethod()	asap2userlib.tlc
/end MODULE	**/end MODULE**	
File footer/tail	%<ASAP2UserFcnWriteFileTail()>	asap2userlib.tlc

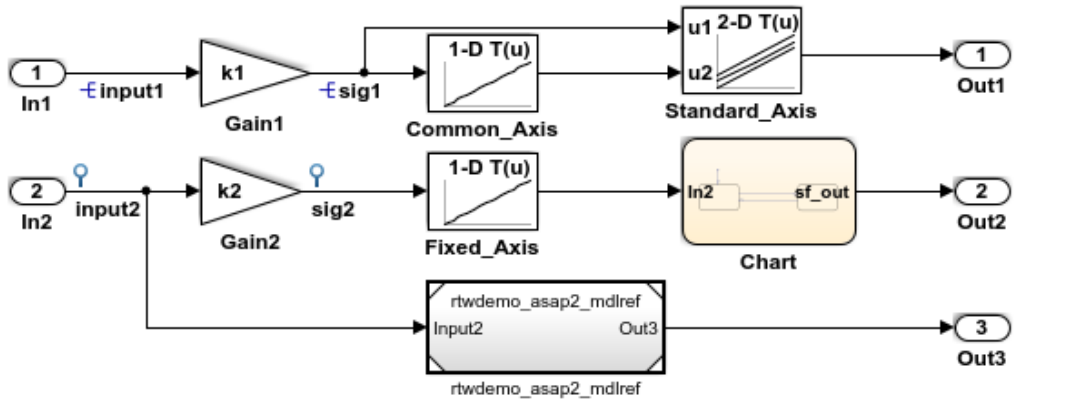
Create a Host-Based ASAM-ASAP2 Data Definition File for Data Measurement and Calibration

This model shows ASAP2 data export. ASAP2 is a data definition standard proposed by the Association for Standardization of Automation and Measuring Systems (ASAM).

Open Example Model

Open the example model `rtwdemo_asap2`.

```
open_system('rtwdemo_asap2');
```



This model shows ASAP2 data export. ASAP2 is a data definition standard proposed by the Association for Standardization of Automation and Measuring Systems (ASAM). ASAP2 is a non-object-oriented description of the data used for measurement, calibration, and diagnostics systems. For more information on ASAM and the ASAP2 standard, see the ASAM Web site: http://www.asam.de.

ASAP2 data definition is achieved with Simulink data objects and test point signals. Using the Target Language Compiler (TLC), you can create highly customized solutions for your application. See the Simulink Coder documentation for details on ASAP2 file generation.

You can configure ASAP2 file generation from "Code Generation > Interface > ASAP2 interface" in the Configuration Parameters dialog box.

Generate Code Using Simulink Coder (double-click) | **Generate Code Using Embedded Coder (double-click)** | **View Interface Configuration (double-click)** | **View Workspace Variables (double-click)**

ASAP2 is a non-object-oriented description of the data used for measurement, calibration, and diagnostics systems. For more information on ASAM and the ASAP2 standard, see the ASAM Web site: http://www.asam.de.

ASAP2 data definition is achieved with Simulink® data objects and test point signals. Using the Target Language Compiler (TLC), you can create highly customized solutions

for your application. See the Simulink Coder® documentation for details on ASAP2 file generation.

You can configure ASAP2 file generation by selecting **ASAP2 interface** on the **Code Generation > Interface** pane of the Configuration Parameters dialog box.

Related Examples

- "Create Tunable Calibration Parameter in the Generated Code" on page 19-60

Direct Memory Access to Generated Code for Simulink Coder

Access Memory in Generated Code Using Global Data Map

Simulink Coder provides a Target Language Compiler (TLC) function library that lets you create a *global data map record*. The global data map record, when generated, is added to the `CompiledModel` structure in the `model.rtw` file. The global data map record is a database containing information required for accessing memory in the generated code, including

- Signals (Block I/O)
- Parameters
- Data type work vectors (DWork)
- External inputs
- External outputs

Use of the global data map requires knowledge of the Target Language Compiler and of the structure of the `model.rtw` file. See "Target Language Compiler Overview" (Simulink Coder) for information on these topics.

The TLC functions that are required to generate and access the global data map record are contained in *`matlabroot`*`/rtw/c/tlc/mw/globalmaplib.tlc`. The comments in the source code fully document the global data map structures and the library functions.

The global data map structures and functions might be modified or enhanced in future releases.

Desktops in Simulink Coder

Accelerate, Refine, and Test Hybrid Dynamic System on Host Computer by Using RSim System Target File

After you create a model, you can use the rapid simulation (RSim) system target file to characterize model behavior. The executable program that results from the build process is for non-real-time execution on your development computer. The executable program is highly optimized for simulating models of hybrid dynamic systems, including models that use variable-step solvers and zero-crossing detection. The speed of the generated code makes the RSim system target file ideal for building programs for batch or Monte Carlo simulations.

About Rapid Simulation

Use the RSim target to generate an executable that runs fast, standalone simulations. You can repeat simulations with varying data sets, interactively or programmatically with scripts, without rebuilding the model. This can accelerate the characterization and tuning of model behavior and code generation testing.

Using command-line options:

- Define parameter values and input signals in one or more MAT-files that you can load and reload at the start of simulations without rebuilding your model.
- Redirect logging data to one or more MAT-files that you can then analyze and compare.
- Control simulation time.
- Specify external mode options.

Note: To run an RSim executable, configure your computer to run MATLAB and have the MATLAB and Simulink installation folders accessible. To deploy a standalone host executable (i.e., without MATLAB and Simulink installed), consider using the Host-Based Shared Library target (ert_shrlib)."

Rapid Simulation Advantage

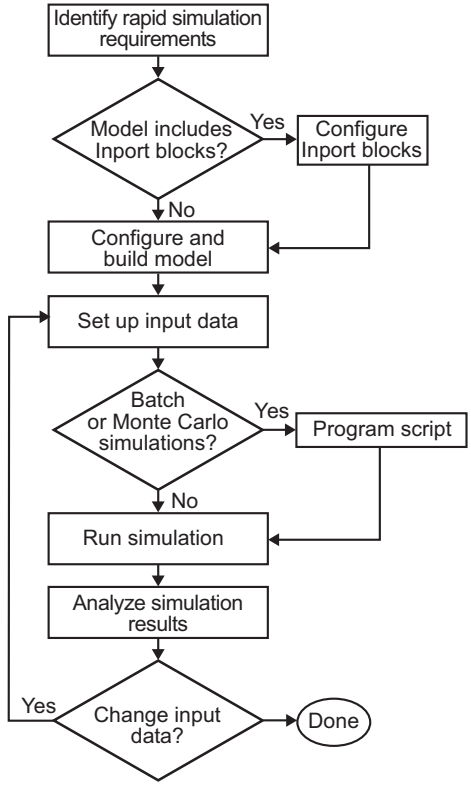
The advantage that you gain from rapid simulation varies. Larger simulations achieve speed improvements of up to 10 times faster than standard Simulink simulations. Some

models might not show noticeable improvement in simulation speed. To determine the speed difference for your model, time your standard Simulink simulation and compare the results with a rapid simulation. In addition, test the model simulation in Rapid Accelerator simulation mode.

General Rapid Simulation Workflow

Like other stages of Model-Based Design, characterization and tuning of model behavior is an iterative process, as shown in the general workflow diagram in the figure. Tasks in the workflow are:

1. Identify your rapid simulation requirements.
2. Configure Inport blocks that provide input source data for rapid simulations.
3. Configure the model for rapid simulation.
4. Set up simulation input data.
5. Run the rapid simulations.



Identify Rapid Simulation Requirements

The first step to setting up a rapid simulation is to identify your simulation requirements.

Question...	For More Information, See...
How long do you want simulations to run?	“Configure and Build Model for Rapid Simulation” on page 46-6
Are there solver requirements? Do you expect to use the same solver for which the model is configured for your rapid simulations?	“Configure and Build Model for Rapid Simulation” on page 46-6

Question...	For More Information, See...
Do your rapid simulations need to accommodate flexible custom code interfacing? Or, do your simulations need to retain storage class settings?	"Configure and Build Model for Rapid Simulation" on page 46-6
Will you be running simulations with multiple data sets?	"Set Up Rapid Simulation Input Data" on page 46-8
Will the input data consist of global parameters, signals, or both?	"Set Up Rapid Simulation Input Data" on page 46-8
What type of source blocks provide input data to the model — From File, Inport, From Workspace?	"Set Up Rapid Simulation Input Data" on page 46-8
Will the model's parameter vector (`model_P`) be used as input data?	"Create a MAT-File That Includes a Model Parameter Structure" on page 46-9
What is the data type of the input parameters and signals?	"Set Up Rapid Simulation Input Data" on page 46-8
Will the source data consist of one variable or multiple variables?	"Set Up Rapid Simulation Input Data" on page 46-8
Does the input data include tunable parameters?	"Create a MAT-File That Includes a Model Parameter Structure" on page 46-9
Do you need to gain access to tunable parameter information — model checksum and parameter data types, identifiers, and complexity?	"Create a MAT-File That Includes a Model Parameter Structure" on page 46-9
Will you have a need to vary the simulation stop time for simulation runs?	"Configure and Build Model for Rapid Simulation" on page 46-6 and"Override a Model Simulation Stop Time" on page 46-21
Do you want to set a time limit for the simulation? Consider setting a time limit if your model experiences frequent zero crossings and has a small minor step size.	"Set a Clock Time Limit for a Rapid Simulation" on page 46-21
Do you need to preserve the output of each simulation run?	"Specify a New Output File Name for a Simulation" on page 46-30 and "Specify New Output File Names for To File Blocks" on page 46-30

Question...	For More Information, See...
Do you expect to run the simulations interactively or in batch mode?	"Scripts for Batch and Monte Carlo Simulations" on page 46-18

Configure Inports to Provide Simulation Source Data

You can use Inport blocks as a source of input data for rapid simulations. To do so, configure the blocks so that they can import data from external MAT-files. By default, the Inport block inherits parameter settings from downstream blocks. In most cases, to import data from an external MAT-file, you must explicitly set the following parameters to match the source data in the MAT-file.

- **Main > Interpolate data**
- **Signal Attributes > Port dimensions**
- **Signal Attributes > Data type**
- **Signal Attributes > Signal type**

If you do not have control over the model content, you might need to modify the data in the MAT-file to conform to what the model expects for input. Input data characteristics and specifications of the Inport block that receives the data must match.

For details on adjusting these parameters and on creating a MAT-file for use with an Inport block, see "Create a MAT-File for an Inport Block" on page 46-14. For descriptions of the preceding block parameters, see the block description of Inport.

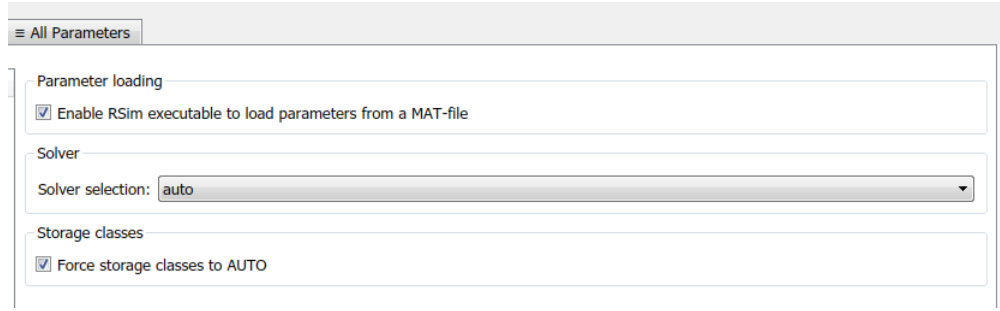
Configure and Build Model for Rapid Simulation

After you identify your rapid simulation requirements, configure the model for rapid simulation.

1. Open the Configuration Parameters dialog box.
2. Go to the **Code Generation** pane.
3. On the **Code Generation** pane, click **Browse**. The System Target File Browser opens.
4. Select `rsim.tlc` (Rapid Simulation Target) and click **OK**.

   On the **Code Generation** pane, the code generator populates the **Make command** and "Template makefile" (Simulink Coder) fields with default settings and adds the **RSim Target** pane under **Code Generation**.

5 Click **RSim Target** to view the **RSim Target** pane.



6 Set the RSim target configuration parameters to your rapid simulation requirements.

If You Want to...	Then...
Generate code that allows the RSim executable to load parameters from a MAT-file	Select **Enable RSim executable to load parameters from a MAT-file** (default).
Let the target choose a solver based on the solver already configured for the model	Set **Solver selection** to `auto` (default). The code generator uses a built-in solver if a fixed-step solver is specified on the **Solver** pane or calls the Simulink solver module (a shared library) if a variable-step solver is specified.
Explicitly instruct the target to use a fixed-step solver	Set **Solver selection** to `Use fixed-step solvers`. In the Configuration Parameters dialog box, on the **Solver** pane, specify a fixed-step solver.
Explicitly instruct the target to use a variable-step solver	Set **Solver selection** to `Use Simulink solver module`. In the Configuration Parameters dialog box, on the **Solver** pane, specify a variable-step solver.
Force storage classes to `Auto` for flexible custom code interfacing	Select **Force storage classes to AUTO** (default).
Retain storage class settings, such as `ExportedGlobal` or `ImportedExtern`, due to application requirements	Clear **Force storage classes to AUTO**.

7 Set up data import and export options. On the **Data Import/Export** pane, in the **Save to Workspace** section, select the **Time**, **States**, **Outputs**, and **Final States** options, as they apply. By default, the code generator saves simulation logging results to a file named *`model`*`.mat`. For more information, see "Export Simulation Data" (Simulink).

8 If you are using external mode communication, set up the interface, using the **Code Generation > Interface** pane. See "What You Can Do with a Host/Target Communication Channel" on page 41-2 for details.

9 Press **Ctrl+B**. The code generator builds a highly optimized executable program that you can run on your development computer with varying data, without rebuilding.

For more information on compilers that are compatible with the Simulink Coder product, see "Select and Configure C or C++ Compiler or IDE" on page 40-3 and "Template Makefiles and Make Options" on page 40-24 .

Set Up Rapid Simulation Input Data

About Rapid Simulation Data Setup

The format and setup of input data for a rapid simulation depends on your requirements.

If the Input Data Source Is...	Then...
The model's global parameter vector (*`model`*`_P`)	Use the `rsimgetrtp` function to get the vector content and then save it to a MAT-file.
The model's global parameter vector and you want a mapping between the vector and tunable parameters	Call the `rsimgetrtp` function to get the global parameter structure and then save it to a MAT-file.
Provided by a From File block	Create a MAT-file that a From File block can read.
Provided by an Inport block	Create a MAT-file that adheres to one of the three data file formats that the Inport block can read.

If the Input Data Source Is...	Then...
Provided by a From Workspace block	Create structure variables in the MATLAB workspace.

The RSim target requires that MAT-files used as input for From File and Inport blocks contain data. The `grt` target inserts MAT-file data directly into the generated code, which is then compiled and linked as an executable. In contrast, RSim allows you to replace data sets for each successive simulation. A MAT-file containing From File or Inport block data must be present if a From File block or Inport block exists in your model.

Create a MAT-File That Includes a Model Parameter Structure

To create a MAT-file that includes a model global parameter structure (*`model`*`_P`),

1. Get the structure by calling the function `rsimgetrtp`.
2. Save the parameter structure to a MAT-file.

If you want to run simulations over varying data sets, consider converting the parameter structure to a cell array and saving the parameter variations to a single MAT-file.

Get the Parameter Structure for a Model

Get the global parameter structure (*`model`*`_P`) for a model by calling the function `rsimgetrtp`.

```
param_struct = rsimgetrtp('model')
```

Argument	Description
`model`	The model for which you are running the rapid simulations.

The `rsimgetrtp` function forces an update diagram action for the specified model and returns a structure that contains the following fields.

Field	Description
`modelChecksum`	A four-element vector that encodes the structure of the model. The code generator uses the checksum to check whether the structure of the model has changed since the RSim executable was generated. If you delete or add a block, and then generate a new *`model`*`_P` vector, the new checksum does not match the

Field	Description
	original checksum anymore. The RSim executable detects this incompatibility in parameter vectors and exits to avoid returning incorrect simulation results. If the model structure changes, you must regenerate the code for the model.
`parameters`	A structure that contains the model's global parameters.

The parameter structure contains the following information.

Field	Description
`dataTypeName`	The name of the parameter data type, for example, `double`
`dataTypeID`	Internal data type identifier used by the code generator
`complex`	The value 0 if real; 1 if complex
dtTransIdx	Internal data index used by the code generator
values	A vector of the parameter values associated with this structure
map	This field contains the mapping information that correlates the 'values' to the tunable parameters of the model. This mapping information, in conjunction with `rsimsetrtpparam`, is useful for creating subsequent rtP structures without compiling the block diagram.

The code generator reports a tunable fixed-point parameter according to its stored value. For example, an `sfix(16)` parameter value of `1.4` with a scaling of `2^-8` has a value of `358` as an `int16`.

In the following example, `rsimgetrtp` returns the parameter structure for the example model rtwdemo_rsimtf to `param_struct`.

```
param_struct = rsimgetrtp('rtwdemo_rsimtf')
```

```
param_struct =

    modelChecksum: [1.7165e+009 3.0726e+009 2.6061e+009 2.3064e+009]
       parameters: [1x1 struct]
```

Save the Parameter Structure to a MAT-File

After you issue a call to `rsimgetrtp`, save the return value of the function call to a MAT-file. Using a command-line option, you can then specify that MAT-file as input for rapid simulations.

The following example saves the parameter structure returned for `rtwdemo_rsimtf` to the MAT-file `myrsimdemo.mat`.

```
save myrsimdemo.mat param_struct;
```

For information on using command-line options to specify required files, see "Run Rapid Simulations" on page 46-18.

Convert the Parameter Structure for Running Simulations on Varying Data Sets

To use rapid simulations to test changes to specific parameters, you can convert the model parameter structure to a cell array. You can then access a specific parameter set by using the @ operator to specify the index for a specific parameter set in the file.

To convert the structure to a cell array:

1. Use the function `rsimgetrtp` to get a structure containing parameter information for the example model `rtwdemo_rsimtf`. Store the structure in a variable `param_struct`.

   ```
   param_struct = rsimgetrtp('rtwdemo_rsimtf');
   ```

   The `parameters` field of the structure is a substructure that contains parameter information. The `values` field of the `parameters` substructure contains the numeric values of the parameters that you can tune during execution of the simulation code.

2. Use the function `rsimsetrtpparam` to expand the structure so that it contains more parameter sets. In this case, create two more parameter sets (for a total of three sets).

   ```
   param_struct = rsimsetrtpparam(param_struct,3);
   ```

   The function converts the `parameters` field to a cell array with three elements. Each element contains information for a single parameter set. By default, the function creates the second and third elements of the cell array by copying the first element. Therefore, all of the parameter sets use the same parameter values.

3. Specify new values for the parameters in the second and third parameter sets.

   ```
   param_struct.parameters{2}.values = [-150 -5000 0 4950];
   param_struct.parameters{3}.values = [-170 -5500 0 5100];
   ```

4. Save the structure containing the parameter set information to a MAT-file.

   ```
   save rtwdemo_rsimtf.mat param_struct;
   ```

Alternatively, you can modify the block parameters in the model, and use `rsimgetrtp` to create multiple parameter sets:

1. Use the function `rsimgetrtp` to get a structure containing parameter information for the example model `rtwdemo_rsimtf`. Store the structure in a variable `param_struct`.

   ```
   param_struct = rsimgetrtp('rtwdemo_rsimtf');
   ```

2. Use the function `rsimsetrtpparam` to expand the structure so that it contains more parameter sets. In this case, create two more parameter sets (for a total of three sets).

   ```
   param_struct = rsimsetrtpparam(param_struct,3);
   ```

   The function converts the `parameters` field to a cell array with three elements. Each element contains information for a single parameter set. By default, the function creates the second and third elements of the cell array by copying the first element. Therefore, all of the parameter sets use the same parameter values.

3. Change the values of block parameters or workspace variables. For example, change the value of the variable `w` from `70` to `72`.

   ```
   w = 72;
   ```

4. Use `rimsgetrtp` to get another structure containing parameter information. Store the structure in a temporary variable `rtp_temp`.

   ```
   rtp_temp = rsimgetrtp('rtwdemo_rsimtf');
   ```

5. Assign the value of the `parameters` field of `rtp_temp` to the structure `param_struct` as a second parameter set.

   ```
   param_struct.parameters{2} = rtp_temp.parameters;
   ```

6. Change the value of the variable `w` from `72` to `75`.

   ```
   w = 75;
   ```

7. Use `rimsgetrtp` to get another structure containing parameter information. Then, assign the value of the `parameters` field to `param_struct` as a third parameter set.

   ```
   rtp_temp = rsimgetrtp('rtwdemo_rsimtf');
   param_struct.parameters{3} = rtp_temp.parameters;
   ```

8. Save the structure containing the parameter set information to a MAT-file.

```
save rtwdemo_rsimtf.mat param_struct;
```

For more information on how to specify each parameter set when you run the simulations, see "Change Block Parameters for an RSim Simulation" on page 46-28.

Create a MAT-File for a From File Block

You can use a MAT-file as the input data source for a From File block. The format of the data in the MAT-file must match the data format expected by that block. For example, if you are using a matrix as an input for the MAT file, this cannot be different from the matrix size for the executable.

To create a MAT-file for a From File block:

1. For array format data, in the workspace create a matrix that consists of two or more rows. The first row must contain monotonically increasing time points. Other rows contain data points that correspond to the time point in that column. The time and data points must be data of type `double`.

   For example:

   ```
   t=[0:0.1:2*pi]';
   Ina1=[2*sin(t) 2*cos(t)];
   Ina2=sin(2*t);
   Ina3=[0.5*sin(3*t) 0.5*cos(3*t)];
   var_matrix=[t Ina1 Ina2 Ina3]';
   ```

   For other supported data types, such as `int16` or fixed-point, the time data points must be of type `double`, just as for array format data. However, the sample data can be of any dimension.

   For more information on setting up the input data, see the block description of From File.

2. Save the matrix to a MAT-file.

   The following example saves the matrix `var_matrix` to the MAT-file `myrsimdemo.mat` in Version 7.3 format.

   ```
   save '-v7.3' myrsimdemo.mat var_matrix;
   ```

   Using a command-line option, you can then specify that MAT-file as input for rapid simulations.

Create a MAT-File for an Inport Block

You can use a MAT-file as the input data source for an Inport block.

The format of the data in the MAT-file must adhere to one of the three column-based formats listed in the following table. The table lists the formats in order from least flexible to most flexible.

Format	Description
Single time/data matrix	• Least flexible. • One variable. • Two or more *columns*. Number of columns must equal the sum of the dimensions of all root Inport blocks plus 1. First column must contain monotonically increasing time points. Other columns contain data points that correspond to the time point in a given row. • Data of type `double`. For an example, see Single time/data matrix in the following procedure, step 4. For more information, see "Create Data Arrays for Root-Level Inports" (Simulink).
Signal-and-time structure	• More flexible than the single time/data matrix format. • One variable. • Must contain two top-level fields: `time` and `signals`. The `time` field contains a *column* vector of the simulation times. The `signals` field contains an array of substructures, each of which corresponds to an Inport block. The substructure index corresponds to the Inport block number. Each `signals` substructure must contain a field named `values`. The `values` field must contain an array of inputs for the corresponding Inport block, where each input corresponds to a time point specified by the `time` field. • If the `time` field is set to an empty value, clear the check box for the Inport block **Interpolate data** parameter. • Data type must match Inport block settings. For an example, see Signal-and-time structure in the following procedure, step 4. For more information on this format, see "Create Data Structures for Root Inports" (Simulink).

Format	Description
Per-port structure	• Most flexible • Multiple variables. Number of variables must equal the number of Inport blocks. • Consists of a separate structure-with-time or structure-without-time for each Inport block. Each Inport block data structure has only one `signals` field. To use this format, in the **Input** text field, enter the names of the structures as a comma-separated list, `in1`, `in2`,..., `inN`, where `in1` is the data for your model's first port, `in2` for the second port, and so on. • Each variable can have a different time vector. • If the `time` field is set to an empty value, clear the check box for the Inport block **Interpolate data** parameter. • Data type must match Inport block settings. • To save multiple variables to the same data file, you must save them in the order expected by the model, using the `-append` option. For an example, see Per-port structure in the following procedure, step 4. For more information, see "Create Data Structures for Root Inports" (Simulink).

The supported formats and the following procedure are illustrated in rtwdemo_rsim_i.

To create a MAT-file for an Inport block:

1. Choose one of the preceding data file formats.
2. Update Inport block parameter settings and specifications to match specifications of the data to be supplied by the MAT-file.

   By default, the Inport block inherits parameter settings from downstream blocks. To import data from an external MAT-file, explicitly set the following parameters to match the source data in the MAT-file.

   - **Main** > **Interpolate data**
   - **Signal Attributes** > **Port dimensions**
   - **Signal Attributes** > **Data type**
   - **Signal Attributes** > **Signal type**

If you choose to use a structure format for workspace variables and the `time` field is empty, you must clear **Interpolate data** or modify the field so that it is set to a nonempty value. Interpolation requires time data.

For descriptions of the preceding block parameters, see the block description of Inport.

3. Build an RSim executable program for the model. The build process creates and calculates a structural checksum for the model and embeds it in the generated executable. The RSim target uses the checksum to verify that data being passed into the model is consistent with what the model executable expects.
4. Create the MAT-file that provides the source data for the rapid simulations. You can create the MAT-file from a workspace variable. Using the specifications in the preceding format comparison table, create the workspace variables for your simulations.

   An example of each format follows:

   Single time/data matrix

```
t=[0:0.1:2*pi]';
Ina1=[2*sin(t) 2*cos(t)];
Ina2=sin(2*t);
Ina3=[0.5*sin(3*t) 0.5*cos(3*t)];
var_matrix=[t Ina1 Ina2 Ina3];
```

   Signal-and-time structure

```
t=[0:0.1:2*pi]';
var_single_struct.time=t;
var_single_struct.signals(1).values(:,1)=2*sin(t);
var_single_struct.signals(1).values(:,2)=2*cos(t);
var_single_struct.signals(2).values=sin(2*t);
var_single_struct.signals(3).values(:,1)=0.5*sin(3*t);
var_single_struct.signals(3).values(:,2)=0.5*cos(3*t);
v=[var_single_struct.signals(1).values...
var_single_struct.signals(2).values...
var_single_struct.signals(3).values];
```

   Per-port structure

```
t=[0:0.1:2*pi]';
Inb1.time=t;
```

```
Inb1.signals.values(:,1)=2*sin(t);
Inb1.signals.values(:,2)=2*cos(t);
t=[0:0.2:2*pi]';
Inb2.time=t;
Inb2.signals.values(:,1)=sin(2*t);
t=[0:0.1:2*pi]';
Inb3.time=t;
Inb3.signals.values(:,1)=0.5*sin(3*t);
Inb3.signals.values(:,2)=0.5*cos(3*t);
```

5 Save the workspace variables to a MAT-file.

Single time/data matrix

The following example saves the workspace variable `var_matrix` to the MAT-file `rsim_i_matrix.mat`.

```
save rsim_i_matrix.mat var_matrix;
```

Signal-and-time structure

The following example saves the workspace structure variable `var_single_struct` to the MAT-file `rsim_i_single_struct.mat`.

```
save rsim_i_single_struct.mat var_single_struct;
```

Per-port structure

To order data when saving per-port structure variables to a single MAT-file, use the `save` command's `-append` option. Be sure to append the data in the order that the model expects it.

The following example saves the workspace variables `Inb1`, `Inb2`, and `Inb3` to MAT-file `rsim_i_multi_struct.mat`.

```
save rsim_i_multi_struct.mat Inb1;
save rsim_i_multi_struct.mat Inb2 -append;
save rsim_i_multi_struct.mat Inb3 -append;
```

The `save` command does not preserve the order in which you specify your workspace variables in the command line when saving data to a MAT-file. For example, if you specify the variables `v1`, `v2`, and `v3`, in that order, the order of the variables in the MAT-file could be `v2 v1 v3`.

Using a command-line option, you can then specify the MAT-files as input for rapid simulations.

Scripts for Batch and Monte Carlo Simulations

The RSim target is for batch simulations in which parameters and input signals vary for multiple simulations. New output file names allow you to run new simulations without overwriting prior simulation results. You can set up a series of simulations to run by creating a `.bat` file for use on a Microsoft Windows platform.

Create a file for the Windows platform with a text editor and execute it by typing the file name, for example, `mybatch`, where the name of the text file is `mybatch.bat`.

```
rtwdemo_rsimtf -f rtwdemo_rsimtf.mat=run1.mat -o results1.mat -tf 10.0
rtwdemo_rsimtf -f rtwdemo_rsimtf.mat=run2.mat -o results2.mat -tf 10.0
rtwdemo_rsimtf -f rtwdemo_rsimtf.mat=run3.mat -o results3.mat -tf 10.0
rtwdemo_rsimtf -f rtwdemo_rsimtf.mat=run4.mat -o results4.mat -tf 10.0
```

In this case, batch simulations run using four sets of input data in files `run1.mat`, `run2.mat`, and so on. The RSim executable saves the data to the files specified with the `-o` option.

The variable names containing simulation results in each of the files are identical. Therefore, loading consecutive sets of data without renaming the data once it is in the MATLAB workspace results in overwriting the prior workspace variable with new data. To avoid overwriting, you can copy the result to a new MATLAB variable before loading the next set of data.

You can also write MATLAB scripts to create new signals and new parameter structures, as well as to save data and perform batch runs using the bang command (`!`).

For details on running simulations and available command-line options, see “Run Rapid Simulations” on page 46-18. For an example of a rapid simulation batch script, see the example “Run Batch Simulations Without Recompiling Generated Code” (Simulink Coder).

Run Rapid Simulations

Rapid Simulations

Using the RSim target, you can build a model once and run multiple simulations to study effects of varying parameter settings and input signals. You can run a simulation directly from your operating system command line, redirect the command from the MATLAB command line by using the bang (!) character, or execute commands from a script.

From the operating system command line, use

```
rtwdemo_rsimtf
```

From the MATLAB command line, use

```
!rtwdemo_rsimtf
```

The following table lists ways you can use RSim target command-line options to control a simulation.

To...	Use...
Read input data for a From File block from a MAT-file other than the MAT-file used for the previous simulation	*model* `-f` *oldfilename*`.mat=`*newfilename*`.mat`
Print a summary of the options for RSim executable targets	*executable filename* `-h`
Read input data for an Inport block from a MAT-file	*model* `-i` *filename*`.mat`
Time out after *n* clock time seconds, where *n* is a positive integer value	*model* `-L` *n*

To...	Use...
Write MAT-file logging data to file *filename*.mat	*model* -o *filename*.mat
Read a parameter vector from file *filename*.mat	*model* -p *filename*.mat
Override the default TCP port (17725) for external mode	*model* -port *TCPport*
Write MAT-file logging data to a MAT-file other than the MAT-file used for the previous simulation	*model* -t *oldfilename*.mat=*newfilename*.mat
Run the simulation until the time value *stoptime* is reached	*model* -tf *stoptime*
Run in verbose mode	*model* -v
Wait for the Simulink engine to start the model in external mode	*model* -w

The following sections use the rtwdemo_rsimtf example model in examples to illustrate some of these command-line options. In each case, the example assumes you have already done the following:

- Created or changed to a working folder.
- Opened the example model.
- Copied the data file *matlabroot*/toolbox/rtw/rtwdemos/rsimdemos/rsim_tfdata.mat to your working folder. You can perform this operation using the command:

  ```
  copyfile(fullfile(matlabroot,'toolbox','rtw','rtwdemos',...
  'rsimdemos','rsim_tfdata.mat'),pwd);
  ```

Requirements for Running Rapid Simulations

The following requirements apply to both fixed and variable step executables.

- You must run the RSim executable on a computer configured to run MATLAB. Also, the RSim.exe file must be able to access the MATLAB and Simulink installation folders on this machine. To obtain that access, your PATH environment variable must include /bin and /bin/($ARCH), where ($ARCH) represents your operating system

architecture. For example, for a personal computer running on a Windows platform, ($ARCH) is “win64”, whereas for a Linux machine, ($ARCH) is “glnxa64”.

- On GNU Linux platforms, to run an RSim executable, define the LD_LIBRARY_PATH environment variable to provide the path to the MATLAB installation folder, as follows:

  ```
  % setenv LD_LIBRARY_PATH /matlab/sys/os/glnx64:$LD_LIBRARY_PATH
  ```

- On the Apple Macintosh OS X platform, to run RSim target executables, you must define the environment variable `DYLD_LIBRARY_PATH` to include the folders `bin/mac` and `sys/os/mac` under the MATLAB installation folder. For example, if your MATLAB installation is under `/MATLAB`, add `/MATLAB/bin/mac` and `/MATLAB/sys/os/mac` to the definition for `DYLD_LIBRARY_PATH`.

Set a Clock Time Limit for a Rapid Simulation

If a model experiences frequent zero crossings and the model's minor step size is small, consider setting a time limit for a rapid simulation. To set a time limit, specify the `-L` option with a positive integer value. The simulation aborts after running for the specified amount of clock time (not simulation time). For example,

```
!rtwdemo_rsimtf -L 20
```

Based on your clock, after the executable runs for 20 seconds, the program terminates. You see a message similar to one of the following:

- On a Microsoft Windows platform,

  ```
  Exiting program, time limit exceeded
  Logging available data ...
  ```

- On The Open Group UNIX platform,

  ```
  ** Received SIGALRM (Alarm) signal @ Fri Jul 25 15:43:23 2003
  ** Exiting model  'vdp' @ Fri Jul 25 15:43:23 2003
  ```

You do not need to do anything to your model or to its configuration to use this option.

Override a Model Simulation Stop Time

By default, a rapid simulation runs until the simulation time reaches the time specified the Configuration Parameters dialog box, on the **Solver** pane. You can override the model simulation stop time by using the `-tf` option. For example, the following simulation runs until the time reaches 6.0 seconds.

```
!rtwdemo_rsimtf -tf 6.0
```

The RSim target stops and logs output data using MAT-file data logging rules.

If the model includes a From File block, the end of the simulation is regulated by the stop time setting specified in the Configuration Parameters dialog box, on the **Solver** pane, or with the RSim target option `-tf`. The values in the block's time vector are ignored. However, if the simulation time exceeds the endpoints of the time and signal matrix (if the final time is greater than the final time value of the data matrix), the signal data is extrapolated to the final time value.

Read the Parameter Vector into a Rapid Simulation

To read the model parameter vector into a rapid simulation, you must first create a MAT-file that includes the parameter structure as described in “Create a MAT-File That Includes a Model Parameter Structure” on page 46-9. You can then specify the MAT-file in the command line with the `-p` option.

For example:

1. Build an RSim executable for the example model `rtwdemo_rsimtf`.
2. Modify parameters in your model and save the parameter structure.

   ```
   param_struct = rsimgetrtp('rtwdemo_rsimtf');
   save myrsimdata.mat param_struct
   ```
3. Run the executable with the new parameter set.

   ```
   !rtwdemo_rsimtf -p myrsimdata.mat

   ** Starting model 'rtwdemo_rsimtf' @ Tue Dec 27 12:30:16 2005
   ** created rtwdemo_rsimtf.mat **
   ```
4. Load workspace variables and plot the simulation results by entering the following commands:

   ```
   load myrsimdata.mat
   plot(rt_yout)
   ```

Specify New Signal Data File for a From File Block

If your model's input data source is a From File block, you can feed the block with input data during simulation from a single MAT-file or you can change the MAT-file from one

simulation to the next. Each MAT-file must adhere to the format described in “Create a MAT-File for a From File Block” on page 46-13.

To change the MAT-file after an initial simulation, you specify the executable with the `-f` option and an *`oldfile`*`.mat=`*`newfile`*`.mat` parameter, as shown in the following example.

1. Set some parameters in the MATLAB workspace. For example:

   ```
   w = 100;
   theta = 0.5;
   ```

2. Build an RSim executable for the example model `rtwdemo_rsimtf`.

3. Run the executable.

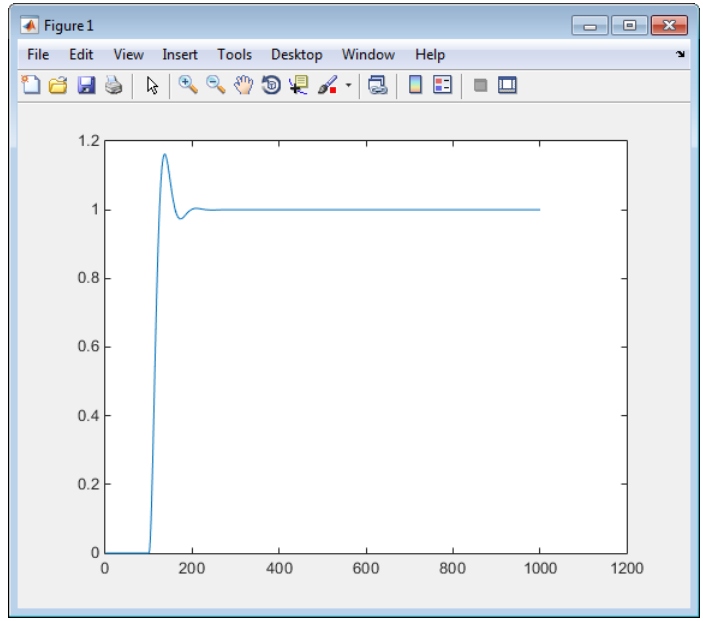
   ```
   !rtwdemo_rsimtf
   ```

   The RSim executable runs a set of simulations and creates output MAT-files containing the specific simulation result.

4. Load the workspace variables and plot the simulation results by entering the following commands:

   ```
   load rtwdemo_rsimtf.mat
   plot(rt_yout)
   ```

   The resulting plot shows simulation results based on default input data.



5 Create a new data file, `newfrom.mat`, that includes the following data:

```
t=[0:.001:1];
u=sin(100*t.*t);
tu=[t;u];
save newfrom.mat tu;
```

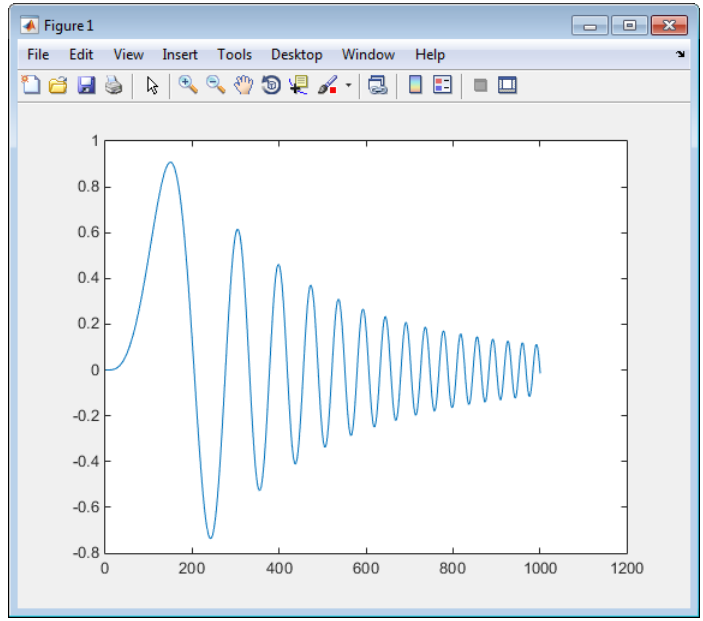
6 Run a rapid simulation with the new data by using the `-f` option to replace the original file, `rsim_tfdata.mat`, with `newfrom.mat`.

```
!rtwdemo_rsimtf -f rsim_tfdata.mat=newfrom.mat
```

7 Load the data and plot the new results by entering the following commands:

```
load rtwdemo_rsimtf.mat
plot(rt_yout)
```

The next figure shows the resulting plot.



From File blocks require input data of type `double`. If you need to import signal data of a data type other than `double`, use an Inport block (see "Create a MAT-File for an Inport Block" on page 46-14) or a From Workspace block with the data specified as a structure.

Workspace data must be in the format:

```
variable.time
variable.signals.values
```

If you have more than one signal, use the following format:

```
variable.time
variable.signals(1).values
variable.signals(2).values
```

Specify Signal Data File for an Inport Block

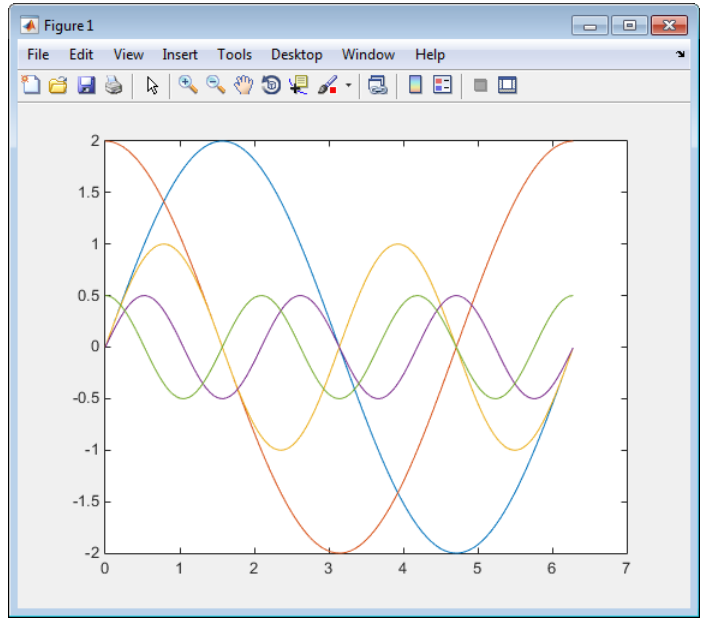
If your model's input data source is an Inport block, you can feed the block with input data during simulation from a single MAT-file or you can change the MAT-file from one

simulation to the next. Each MAT-file must adhere to one of the three formats described in “Create a MAT-File for an Inport Block” on page 46-14.

To specify the MAT-file after a simulation, you specify the executable with the `-i` option and the name of the MAT-file that contains the input data. For example:

1. Open the model rtwdemo_rsim_i.
2. Check the Inport block parameter settings. The following Inport block data parameter settings and specifications that you specify for the workspace variables must match settings in the MAT-file, as indicated in “Configure Inports to Provide Simulation Source Data” on page 46-6:
   - **Main > Interpolate data**
   - **Signal Attributes > Port dimensions**
   - **Signal Attributes > Data type**
   - **Signal Attributes > Signal type**
3. Build the model.
4. Set up the input signals. For example:

```
t=[0:0.01:2*pi]';
s1=[2*sin(t) 2*cos(t)];
s2=sin(2*t);
s3=[0.5*sin(3*t) 0.5*cos(3*t)];
plot(t, [s1 s2 s3])
```



5 Prepare the MAT-file by using one of the three available file formats described in "Create a MAT-File for an Inport Block" on page 46-14. The following example defines a signal-and-time structure in the workspace and names it `var_single_struct`.

```
t=[0:0.1:2*pi]';
var_single_struct.time=t;
var_single_struct.signals(1).values(:,1)=2*sin(t);
var_single_struct.signals(1).values(:,2)=2*cos(t);
var_single_struct.signals(2).values=sin(2*t);
var_single_struct.signals(3).values(:,1)=0.5*sin(3*t);
var_single_struct.signals(3).values(:,2)=0.5*cos(3*t);
v=[var_single_struct.signals(1).values...
var_single_struct.signals(2).values...
var_single_struct.signals(3).values];
```

6 Save the workspace variable `var_single_struct` to MAT-file `rsim_i_single_struct`.

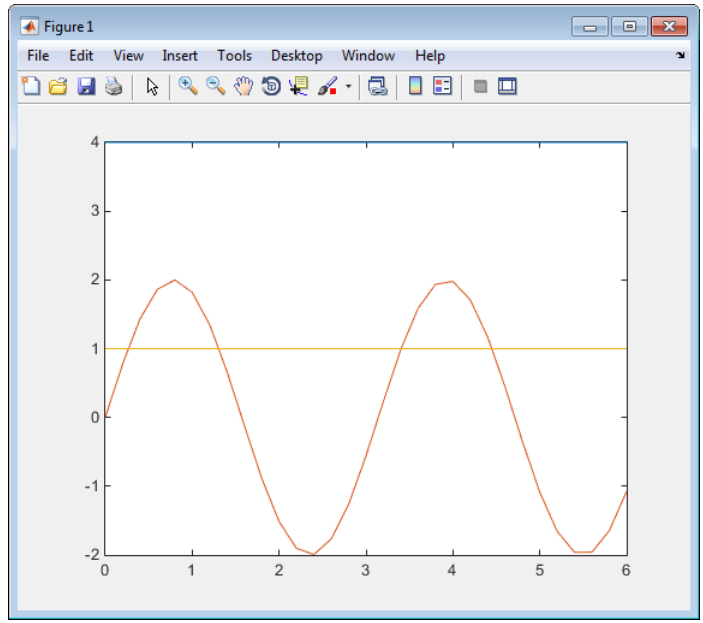
```
save rsim_i_single_struct.mat var_single_struct;
```

7 Run a rapid simulation with the input data by using the `-i` option. Load and plot the results.

```
!rtwdemo_rsim_i -i rsim_i_single_struct.mat

** Starting model 'rtwdemo_rsim_i' @ Tue Aug 19 10:26:53 2014
 *** rsim_i_single_struct.mat is successfully loaded! ***
** created rtwdemo_rsim_i.mat **

** Execution time = 0.02024185130718954s
```

8 Load and plot the results.

```
load rtwdemo_rsim_i.mat
plot(rt_tout, rt_yout);
```



Change Block Parameters for an RSim Simulation

As described in "Create a MAT-File That Includes a Model Parameter Structure" on page 46-9, after you alter one or more parameters in a Simulink block diagram, you can extract the parameter vector, *`model`*`_P`, for the entire model. You can then save the parameter vector, along with a model checksum, to a MAT-file. This MAT-file can be read directly by the standalone RSim executable, allowing you to replace the entire parameter

vector or individual parameter values, for running studies of variations of parameter values representing coefficients, new data for input signals, and so on.

RSim can read the MAT-file and replace the entire *model*_P structure whenever you change one or more parameters, without recompiling the entire model.

For example, assume that you changed one or more parameters in your model, generated the new *model*_P vector, and saved *model*_P to a new MAT-file called `mymatfile.mat`. To run the same `rtwdemo_rsimtf` model and use these new parameter values, use the `-p` option, as shown in the following example:

```
!rtwdemo_rsimtf -p mymatfile.mat
load rtwdemo_rsimtf
plot(rt_yout)
```

If you have converted the parameter structure to a cell array for running simulations on varying data sets, as described in "Convert the Parameter Structure for Running Simulations on Varying Data Sets" on page 46-11, you must add an `@`*n* suffix to the MAT-file specification. *n* is the element of the cell array that contains the specific input that you want to use for the simulation.

The following example converts `param_struct` to a cell array, changes parameter values, saves the changes to MAT-file `mymatfile.mat`, and then runs the executable using the parameter values in the second element of the cell array as input.

```
param_struct = rsimgetrtp('rtwdemo_rsimtf');
param_struct = rsimsetrtpparam(param_struct,2);
param_struct.parameters{1}

ans =

       dataTypeName: 'double'
         dataTypeId: 0
            complex: 0
         dtTransIdx: 0
             values: [-140 -4900 0 4900]
                map: []
    structParamInfo: []

param_struct.parameters{2}.values=[-150 -5000 0 4950];
save mymatfile.mat param_struct;
!rtwdemo_rsimtf -p mymatfile.mat@2 -o rsim2.mat
```

Specify a New Output File Name for a Simulation

If you have specified one or more of the **Save to Workspace** options — **Time**, **States**, **Outputs**, or **Final States** — in the Configuration Parameters dialog box, on the **Data Import/Export** pane, the default is to save simulation logging results to the file *`model`*`.mat`. For example, the example model `rtwdemo_rsimtf` normally saves data to `rtwdemo_rsimtf.mat`, as follows:

```
!rtwdemo_rsimtf
created rtwdemo_rsimtf.mat
```

You can specify a new output file name for data logging by using the `-o` option when you run an executable.

```
!rtwdemo_rsimtf -o rsim1.mat
```

In this case, the set of parameters provided at the time of code generation, including From File block data parameters, is run.

Specify New Output File Names for To File Blocks

In much the same way as you can specify a new system output file name, you can also provide new output file names for data saved from one or more To File blocks. To do this, specify the original file name at the time of code generation with a new name, as shown in the following example:

```
!rtwdemo_rsimtf -t rtwdemo_rsimtf_data.mat=mynewrsimdata.mat
```

In this case, assume that the original model wrote data to the output file `rtwdemo_rsimtf_data.mat`. Specifying a new file name forces RSim to write to the file `mynewrsimdata.mat`. With this technique, you can avoid overwriting an existing simulation run.

Tune Parameters Interactively During Rapid Simulation

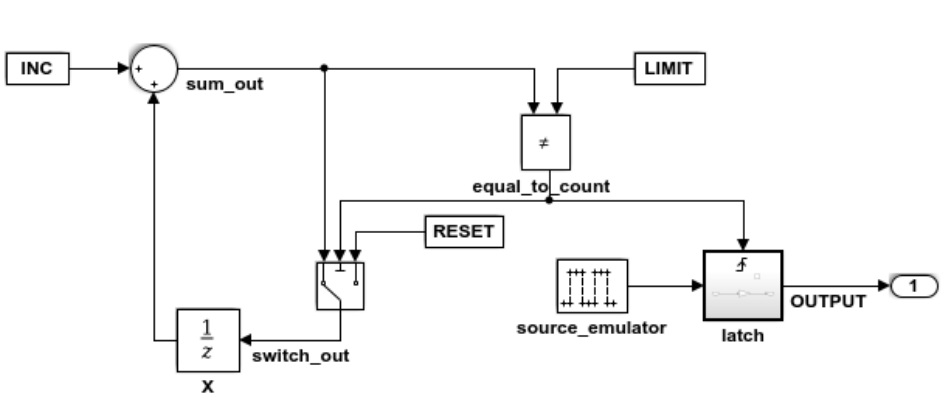
The RSim target was designed to let you run batch simulations at the fastest possible speed. Using variable-step or fixed-step solvers with RSim combined with the use of a tunable parameter data structure, whether you set **Default parameter behavior** to `Tunable` or to `Inlined`, you can create multiple parameter sets to run with the RSim target's standalone executable file (.exe on Windows) generated using Simulink Coder. Each invocation of the executable allows specification of the file name to use for results.

For this example, **Default parameter behavior** is set to `Inlined`. The model declares workspace variables as tunable parameters. To use RSim with **Default parameter behavior** set to `Tunable`, and without explicitly declaring tunable parameters, see "Run Batch Simulations Without Recompiling Generated Code" (Simulink Coder).

Open Example Model

Open the example model `rtwdemo_rsim_param_tuning`.

```
open_system('rtwdemo_rsim_param_tuning');
```



The RSim target was designed to let you run batch simulations at the fastest possible speed. Using variable-step or fixed-step solvers with RSim combined with the use of a tunable parameter data structure, you can create multiple parameter sets to run with the RSim target's standalone executable file (.exe on Windows) generated using Simulink Coder. Each invocation of the executable allows specification of the filename to use for results.

This model uses the RSim target to allow a non-real-time executable to be passed new data without the need to recompile the Simulink model. This feature allows you to easily get a map of the tunable parameters declared in a model and save it in a MAT-file. You can then create your own MATLAB GUI or a standalone GUI (independent of MATLAB) to read and write the MAT-file and rerun the executable to produce new output files.

View MATLAB programs

Double-click the buttons at the upper right sequentially to run the example. To review the code used to create both the MATLAB GUI and standalone GUI, double-click the View MATLAB programs button.

For more information, you can also refer to the "Rapid Simulation Target" section in the Simulink Coder documentation.

This model uses the RSim target and the `rsimgetrtp` function to allow a non real time executable to be passed new data without the need to recompile the Simulink model. This feature allows you to get a map of the tunable parameters declared in a model and save it in a MAT-file. You can then create your own MATLAB GUI or a standalone GUI (independent of MATLAB) to read and write the MAT-file and rerun the executable to produce new output files.

Double-click the buttons at the upper right sequentially to run the example.

To review the code used to create both the MATLAB GUI and standalone GUI, double-click the View MATLAB programs button.

For more information, you can also refer to the "Rapid Simulation Target" section in the Simulink Coder documentation.

Rapid Simulation Target Limitations

The RSim target has the following limitations:

- Does not support algebraic loops.
- Does not support Interpreted MATLAB Function blocks.
- Does not support noninlined MATLAB language or Fortran S-functions.
- If an RSim build includes referenced models (by using Model blocks), set up these models to use fixed-step solvers to generate code for them. The top model, however, can use a variable-step solver as long as the blocks in the referenced models are discrete.
- In certain cases, changing block parameters can result in structural changes to your model that change the model checksum. An example of such a change is changing the number of delays in a DSP simulation. In such cases, you must regenerate the code for the model.

More About

- “Acceleration” (Simulink)
- “Accelerate Simulation, Reuse Code, or Protect Intellectual Property by Using S-Function Target” (Simulink Coder)

Accelerate Simulation, Reuse Code, or Protect Intellectual Property by Using S-Function Target

S-functions are an important class of system target file for which the code generator can produce code. The ability to encapsulate a subsystem into an S-function allows you to increase its execution efficiency and facilitate code reuse.

The following sections describe the properties of S-function targets and illustrate how to generate them. For more details on the structure of S-functions, see "Host-Specific Code" (Simulink).

In this section...

About the S-Function Target

Using the S-function target, you can build an S-function component and use it as an S-Function block in another model. The following sections describe deployment considerations for the S-function target.

- "Required Files for S-Function Deployment" on page 46-35
- "Sample Time Propagation in Generated S-Functions" on page 46-36
- "Choose a Solver Type" on page 46-36
- "Solver Type Overrides" on page 46-37

The `'S-Function'` value for `CodeFormat` TLC variable used by the S-function target generates code that conforms to the Simulink C MEX S-function application programming interface (API). Applications of this format include

- Conversion of a model to a component. You can generate an S-Function block for a model, `m1`. Then, you can place the generated S-Function block in another model, `m2`. Regenerating code for `m2` does not require regenerating code for `m1`.
- Conversion of a subsystem to a component. By extracting a subsystem to a separate model and generating an S-Function block from that model, you can create a reusable component from the subsystem. See "Create S-Function Blocks from a Subsystem" on page 46-37 for an example of this procedure.
- Speeding up simulation. Often, an S-function generated from a model performs more efficiently than the original model.
- Code reuse. You can incorporate multiple instances of one model inside another without replicating the code for each instance. Each instance continues to maintain its own unique data.

You can place a generated S-function block into another model from which you can generate another S-function. This approach allows any level of nested S-functions. For limitations related to nesting, see "Limitations on Nesting S-Functions" on page 46-49.

Note: While the S-function target provides a means to deploy an application component for reuse while shielding its internal logic from inspection and modification, the preferred solutions for protecting intellectual property in distributed components are:

- The protected model, a referenced model that hides all block and line information. For more information, see "Protected Model" (Simulink).
- The shared library system target file, used to generate a shared library for a model or subsystem for use in a system simulation external to Simulink. For more information, see "Package Generated Code as Shared Libraries" on page 47-2.

Required Files for S-Function Deployment

There are different files required to deploy a generated S-Function block for simulation versus code generation.

To deploy your generated S-Function block for inclusion in other models *for simulation*, you need only provide the binary MEX-file object that was generated in the current working folder when the S-Function block was created. The required file is:

- *subsys*`_sf.`*mexext*

where *subsys* is the subsystem name and *mexext* is a platform-dependent MEX-file extension (see mexext (MATLAB)). For example, `SourceSubsys_sf.mexw64`.

To deploy your generated S-Function block for inclusion in other models *for code generation*, you must provide all of the files that were generated in the current working folder when the S-Function block was created. The required files are:

- *subsys*`_sf.c` or `.cpp`, where *subsys* is the subsystem name (for example, `SourceSubsys_sf.c`)
- *subsys*`_sf.h`
- *subsys*`_sf.`*mexext*, where *mexext* is a platform-dependent MEX-file extension (see mexext (MATLAB))
- Subfolder *subsys*`_sfcn_rtw` and its contents

Note: The generated S-function code uses **Configuration Parameters > Hardware Implementation** parameter values that match the host system on which the function was built. When you use the S-function in a model for code generation, make sure that these parameter values for the model match the parameter values of the S-function.

Sample Time Propagation in Generated S-Functions

A generated S-Function block can inherit its sample time from the model in which it is placed if certain criteria are met. Conditions that govern sample time propagation for both Model blocks and generated S-Function blocks are described in "Sample Times for Model Referencing" (Simulink) and "Inherited Sample Time for Referenced Models" on page 5-23.

To generate an S-Function block that meets the criteria for inheriting sample time, you must constrain the solver for the model from which the S-Function block is generated. On the **Solver** configuration parameters dialog box pane, set **Type** to `Fixed-step` and **Periodic sample time constraint** to `Ensure sample time independent`. If the model is unable to inherit sample times, this setting causes the Simulink software to display an error message when building the model. For more information about this option, see "Periodic sample time constraint" (Simulink).

Choose a Solver Type

The table shows the possible combinations of top-level model solver types as these types relate to subsystem build types and solver types for generated S-functions.

Top-level Model Solver Options and Subsystem Build Types

	Model Configuration Parameters: top-level model configuration	
Subsystem Build Type	**Solver options, Type: Variable-step**	**Solver options, Type: Fixed-step**
Build This Subsystem	Generated S-function does NOT require variable-step solver	Generated S-function does NOT require variable-step solver
Generate S-Function	Generated S-function requires variable-step solver	Generated S-function does NOT require variable-step solver

Note: S-functions generated from a subsystem have parameters that are hard coded into the block. Simulink calculates parameters such as sample time when it generates the block, not during simulation run time. Hence, it is important to verify whether the generated S-Function block works as expected in the destination model.

Solver Type Overrides

There are instances when the subsystem build type selection produces an override of the subsystem solver type. The table summarizes the relationships between subsystem build types and the applied subsystem solver types.

Top-level Model Solver Type Overrides of Subsystem Solver Types by Build Type

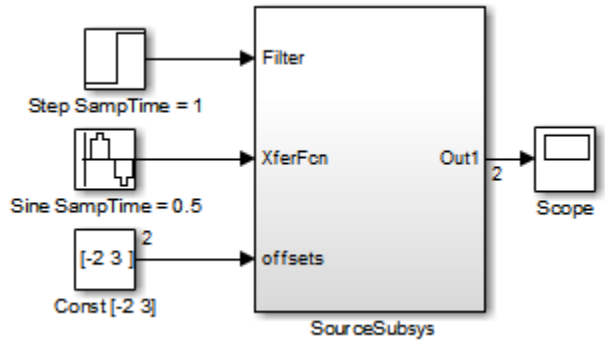
	Model Configuration Parameters: top-level model configuration	
Subsystem Build Type	**Solver options, Type: Variable-step**	**Solver options, Type: Fixed-step**
Build This Subsystem	Subsystem uses fixed-step solver type	Subsystem uses fixed-step solver type
Generate S-Function	Subsystem uses variable-step solver type	Subsystem uses fixed-step solver type

Create S-Function Blocks from a Subsystem

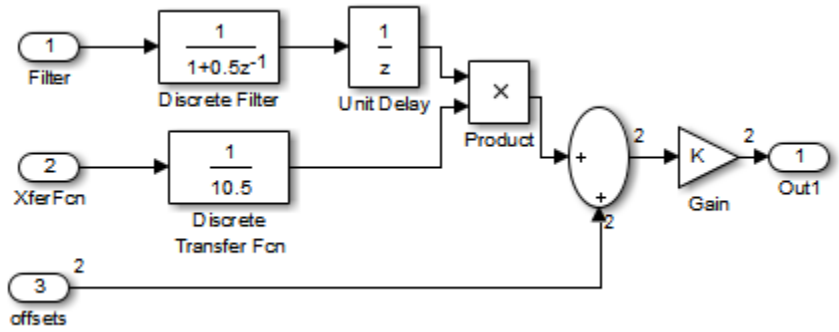
This section illustrates how to extract a subsystem from a model and generate a reusable S-function component from it.

The next figure shows `SourceModel`, a simple model that inputs signals to a subsystem. The subsequent figure shows the subsystem, `SourceSubsys`. The signals, which have different widths and sample times, are:

- A Step block with sample time 1
- A Sine Wave block with sample time 0.5
- A Constant block whose value is the vector [-2 3]



SourceModel



SourceSubsys

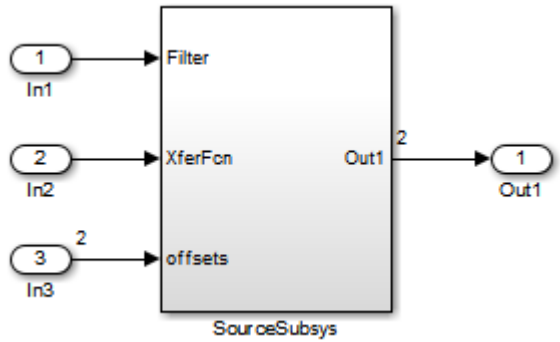
The objective is to extract `SourceSubsys` from the model and build an S-Function block from it, using the S-function target. The S-Function block must perform identically to the subsystem from which it was generated.

In this model, `SourceSubsys` inherits sample times and signal widths from its input signals. However, S-Function blocks created from a model using the S-function target has all signal attributes (such as signal widths or sample times) hard-wired. (The sole exception to this rule concerns sample times, as described in "Sample Time Propagation in Generated S-Functions" on page 46-36.)

In this example, you want the S-Function block to retain the properties of `SourceSubsys` as it exists in `SourceModel`. Therefore, before you build the subsystem as a separate S-function component, you must set the inport sample times and widths explicitly. In addition, the solver parameters of the S-function component must be the same as those parameters of the original model. The generated S-function component operates identically to the original subsystem (see "Choose a Solver Type" on page 46-36 for more information).

To build `SourceSubsys` as an S-function component,

1. Create a new model and copy/paste the `SourceSubsys` block into the empty window.
2. Set the signal widths and sample times of inports inside `SourceSubsys` such that they match those of the signals in the original model. Inport 1, `Filter`, has a width of 1 and a sample time of 1. Inport 2, `Xferfcn`, has a width of 1 and a sample time of 0.5. Inport 3, `offsets`, has a width of 2 and a sample time of 0.5.
3. The generated S-Function block should have three inports and one outport. Connect inports and an outport to `SourceSubsys`, as shown in the next figure.



The signal widths and sample times are propagated to these ports.

4 Set the solver type, mode, and other solver parameters such that they are identical to those of the source model. This is easiest to do if you use Model Explorer.

5 In the Configuration Parameters dialog box, go to the **Code Generation** pane.

6 Click **Browse** to open the System Target File Browser.

7 In the System Target File Browser, select the S-function target, `rtwsfcn.tlc`, and click **OK**.

8 Select the **S-Function Target** pane. Make sure that **Create new model** is selected, as shown in the next figure:

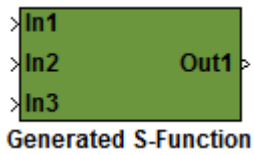
When this option is selected, the build process creates a new model after it builds the S-function component. The new model contains an S-Function block, linked to the S-function component.

Click **Apply**.

9 Save the new model containing your subsystem, for example as `SourceSubsys`.

10 Build the model.

11 The build process produces the S-function component in the working folder. After the build, a new model window is displayed.

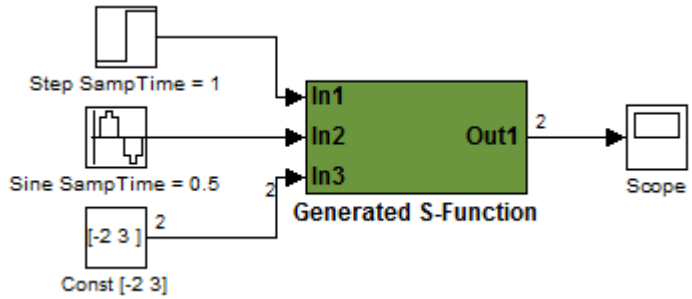


Optionally you can save the generated model, for example as `SourceSubsys_Sfunction`.

12 You can now copy the S-Function block generated from the new model and use it in other models or in a library.

Note: For a list of files required to deploy your S-Function block for simulation or code generation, see "Required Files for S-Function Deployment" on page 46-35.

The next figure shows the S-Function block plugged into the original model. Given identical input signals, the S-Function block performs identically to the original subsystem.



Generated S-Function Configured Like SourceModel

The speed at which the S-Function block executes is typically faster than the original model. This difference in speed is more pronounced for larger and more complicated models. By using generated S-functions, you can increase the efficiency of your modeling process.

Tunable Parameters in Generated S-Functions

You can use tunable parameters in generated S-functions in two ways:

- Use the **Generate S-function** feature (see "Automate S-Function Generation with S-Function Builder" on page 11-61).

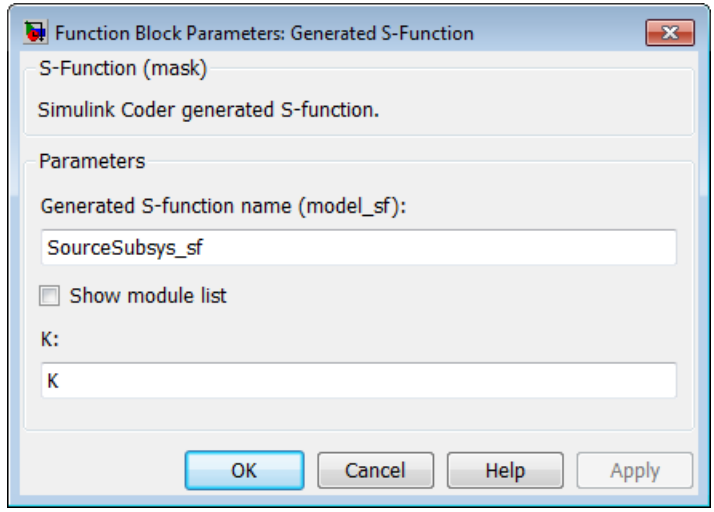
  or

- Use the Model Parameter Configuration dialog box (see "Block Parameter Representation in the Generated Code" on page 19-47) to declare desired block parameters tunable.

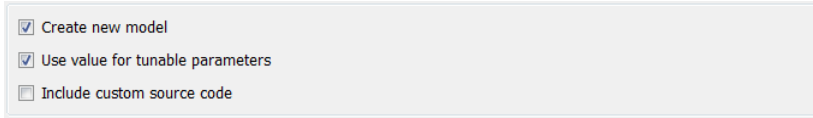
Block parameters that are declared tunable with the `auto` storage class in the source model become tunable parameters of the generated S-function. These parameters do not become part of a generated *`model`*`_P` (formerly `rtP`) parameter data structure, as they would in code generated from other targets. Instead, the generated code accesses these parameters by using MEX API calls such as `mxGetPr` or `mxGetData`. Your code should access these parameters in the same way.

For more information on MEX API calls, see "About C S-Functions" (Simulink) and "MATLAB API for Other Languages" (MATLAB).

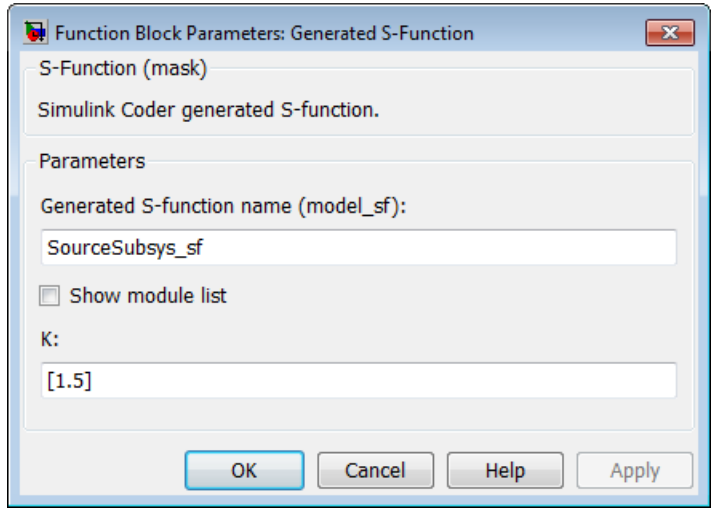
S-Function blocks created by using the S-function target are automatically masked. The mask displays each tunable parameter in an edit field. By default, the edit field displays the parameter by variable name, as in the following example.



You can choose to display the value of the parameter rather than its variable name by selecting **Use value for tunable parameters** on the **Code Generation > S-Function Target** pane of the Configuration Parameters dialog box.



When this option is chosen, the value of the variable (at code generation time) is displayed in the edit field, as in the following example.



System Target File

The `rtwsfcn.tlc` system target file is provided for use with the S-function target.

Checksums and the S-Function Target

The code generator creates a checksum for a model and uses the checksum during the build process for code reuse, model reference, and external mode features.

The code generator calculates a model checksum by

1. Calculating a checksum for each subsystem in the model. A subsystem's checksum is the combination of properties (data type, complexity, sample time, port dimensions, and so forth) of the subsystem's blocks.
2. Combining the subsystem checksums and other model-level information.

An S-function can add additional information, not captured during the block property analysis, to a checksum by calling the function `ssSetChecksumVal`. For the S-Function target, the value that gets added to the checksum is the checksum of the model or subsystem from which the S-function is generated.

The code generator applies the subsystem and model checksums as follows:

- Code reuse — If two subsystems in a model have the same checksum, the code generator produces code for one function only.
- Model reference — If the current model checksum matches the checksum when the model was built, the build process does not rebuild referenced models.
- External mode — If the current model checksum does not match the checksum of the code that is running on the target hardware, the build process generates an error.

Generated S-Function Compatibility

When you build a MEX S-function from your model, the code generator builds a level 2 noninlined S-function. Cross-release usage limitations on the generated code and binary MEX file (for example, `*.mexw64`) include:

- S-function target generated code from previous MATLAB release software is not compatible with newer releases. Do not recompile the generated code from a previous release with newer MATLAB release software. Use the same MATLAB release software to generate code for the S-function target and compile the code into a MEX file.
- You can use binary S-function MEX files generated from previous MATLAB release software with the same or newer releases with the same compatibility considerations as handwritten S-functions. For more information, see "S-Function Compatibility" (Simulink).
- The code generator can generate code and build an executable from a model that contains generated S-functions. This support requires that the S-functions are built with the same MATLAB release software that builds the model. It is not possible to incorporate a generated S-function MEX file from previous MATLAB release software into a model and build the model with newer releases.

S-Function Target Limitations

Limitations on Using Tunable Variables in Expressions

Certain limitations apply to the use of tunable variables in expressions. When the code generator encounters an unsupported expression while producing code, a warning appears and the equivalent numeric value is generated in the code. For a list of the limitations, see "Tunable Expression Limitations" on page 19-53.

Parameter Tuning

The S-Function block does not support tuning of tunable parameters with:

- Complex values.
- Values or data types that are transformed to a constant (by setting the model configuration parameter **Optimization > Signals and Parameters > Default parameter behavior** to `Inlined`).
- Data types that are not built-in.

If you select these tunable parameters (through the Generate S-Function for Subsystem dialog box):

- The software produces warnings during the build process.
- The generated S-Function block mask does not display these parameters.

Run-Time Parameters and S-Function Compatibility Diagnostics

If you set the **S-function upgrades needed** option on the **Diagnostics > Compatibility** pane of the Configuration Parameters dialog box to `warning` or `error`, the code generator instructs you to upgrade S-functions that you create with the **Generate S-function** feature. This is because the S-function system target file does

not register run-time parameters. Run-time parameters are only supported for inlined S-Functions and the generated S-Function supports features that prevent it from being inlined (for example, it can call or contain other noninlined S-functions).

You can work around this limitation by setting the **S-function upgrades needed** option to `none`.

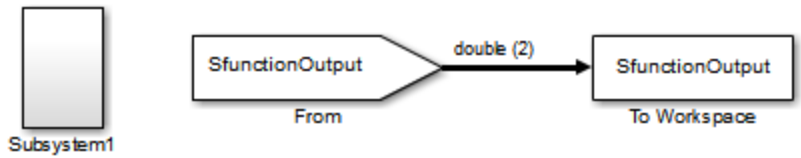
Limitations on Using Goto and From Block

When using the S-function system target file, the code generator restricts I/O to correspond to the root model Inport and Outport blocks (or the Inport and Outport blocks of the Subsystem block from which the S-function target was generated). No code is generated for Goto or From blocks.

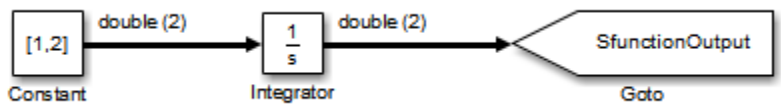
To work around this restriction, create your model and subsystem with the required Inport and Outport blocks, instead of using Goto and From blocks to pass data between the root model and subsystem. In the model that incorporates the generated S-function, you would then add Goto and From blocks.

Example Before Work Around

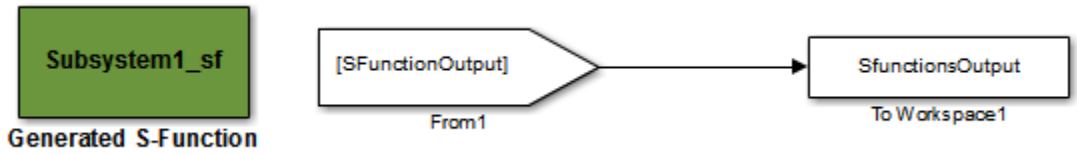
- Root model with a From block and subsystem, `Subsystem1`



- `Subsystem1` with a Goto block, which has global visibility and passes its input to the From block in the root model

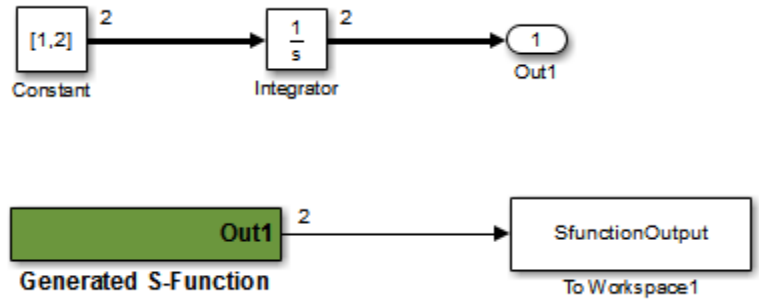


- `Subsystem1` replaced with an S-function generated with the S-Function target — a warning results when you run the model because the generated S-function does not implement the Goto block



Example After Work Around

An Outport block replaces the GoTo block in `Subsystem1`. When you plug the generated S-function into the root model, its output connects directly to the To Workspace block.



Limitations on Building and Updating S-Functions

The following limitations apply to building and updating S-functions using the S-function system target file:

- You cannot build models that contain Model blocks using the S-function system target file. This also means that you cannot build a subsystem module by right-clicking (or by using **Code > C/C++ Code > Build Selected Subsystem**) if the subsystem contains Model blocks. This restriction applies only to S-functions generated using the S-function target, not to ERT S-functions.
- If you modify the model that generated an S-Function block, the build process does not automatically rebuild models containing the generated S-Function block. This is in contrast to the practice of automatically rebuilding models referenced by Model blocks when they are modified (depending on the Model Reference **Rebuild** configuration setting).

- Handwritten S-functions without corresponding TLC files must contain exception-free code. For more information on exception-free code, see "Exception Free Code" (Simulink).

Unsupported Blocks

The S-function format does not support the following built-in blocks:

- Interpreted MATLAB Function block
- S-Function blocks containing any of the following:
  - MATLAB language S-functions (unless you supply a TLC file for C code generation)
  - Fortran S-functions (unless you supply a TLC file for C code generation)
  - C/C++ MEX S-functions that call into the MATLAB environment
- Scope block
- To Workspace block

The S-function format does not support blocks from the `embeddedtargetslib` block library.

`SimState` Not Supported for Code Generation

You can use `SimState` within C-MEX and Level-2 MATLAB language S-functions to save and restore the simulation state. See "S-Function Compliance with the SimState" (Simulink). However, `SimState` is not supported for code generation, including with the S-function system target file.

Profiling Code Performance with TLC Hook Function Not Supported

Profiling the performance of generated code using the Target Language Compiler (TLC) hook function interface described in "Profile Code Performance" (Simulink Coder) is not supported for the S-function target.

Note: If you have an Embedded Coder license, see "Code Execution Profiling" for an alternative and simpler approach based on software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations.

Limitations on Nesting S-Functions

The following limitations apply to nesting a generated S-Function block in a model or subsystem from which you generate another S-function:

- The software does not support nonvirtual bus input and output signals for a nested S-function.
- You should avoid nesting an S-function in a model or subsystem having the same name as the S-function (possibly several levels apart). In such situations, the S-function can be called recursively. The software currently does not detect such loops in S-function dependency, which can result in aborting or hanging your MATLAB session. To prevent this from happening, be sure to name the subsystem or model to be generated as an S-function target uniquely, to avoid duplicating existing MEX filenames on the MATLAB path.

Limitations on User-Defined Data Types

The S-function system target file does not support the `HeaderFile` property that can be specified on user-defined data types, including those based on `Simulink.AliasType`, `Simulink.Bus`, and `Simulink.NumericType` objects. If a user-defined data type in your model uses the `HeaderFile` property to specify an associated header file, code generation with the S-function system target file disregards the value and does not generate a corresponding include statement.

Limitation on Right-Click Generation of an S-Function Target

If you generate an S-function target by right-clicking a Function-Call Subsystem block, the original subsystem and the generated S-function might not be consistent. An inconsistency occurs when the **States when enabling** parameter of the Trigger Port block inside the Function-Call Subsystem block is set to **inherit**. You must set the **States when enabling** parameter to **reset** or **held**, otherwise Simulink reports an error.

Limitation on S-Functions with Bus I/O Signals

If an S-function generated using the S-function target has bus input or output signals, the generated bus data structures might include padding to align fields of the bus elements with the Simulink representation used during simulation. However, if you insert the S-function in a model and generate code using a model target such as `grt.tlc`, the bus structure alignment generated for the model build might be incompatible with the padding generated for the S-function and might affect the

numerical results of code execution. To make the structure alignment consistent between model simulation and execution of the model code, for each `Simulink.Bus` object, you can modify the `HeaderFile` property to remove the unpadded bus structure header file. This will cause the bus typedefs generated for the S-function to be reused in the model code.

Limitation on Subsystems with Function-Call I/O Signals

The S-function target does not support creating an S-Function block from a subsystem that has a function-call trigger input or a function-call output.

Data Store Access

When an S-Function in your model accesses a data store during simulation, Simulink disables data store diagnostics.

- If you created the S-Function from a model, the diagnostic is disabled for global data stores as well.
- If you created the S-Function from a subsystem, the diagnostic is disabled for the following data stores:
  - Global data stores
  - Data stores placed outside the subsystem, but accessed by Data Store Read or Data Store Write blocks.

Cannot Specify Outport Dimensions Through Subsystem Mask

You cannot specify **Port dimensions** for an Outport block through a subsystem mask if you want to generate an S-Function block from the subsystem. The software produces an error when you try to run a simulation that uses the S-Function block, for example:

```
Invalid setting in 'testSystem/Subsystem/__OutputSSForSFun__/Out2'
for parameter 'PortDimensions'
...
```

More About

- “Acceleration” (Simulink)
- “Accelerate, Refine, and Test Hybrid Dynamic System on Host Computer by Using RSim System Target File” (Simulink Coder)

47

Desktops in Embedded Coder

Package Generated Code as Shared Libraries

If you have an Embedded Coder license, you can package generated source code from a model component for easy distribution and shared use by building the code as a shared library—Windows dynamic link library (`.dll`), UNIX shared object (`.so`), or Macintosh OS X dynamic library (`.dylib`). You or others can integrate the shared library into an application that runs on a Windows, UNIX, or Macintosh OS X development computer. The generated `.dll`, `.so`, or `.dylib` file is shareable among different applications and upgradeable without having to recompile the applications that use it.

About Generated Shared Libraries

You build a shared library by configuring the code generator to use the system target file `ert_shrlib.tlc`. Code generation for that system target file 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:

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

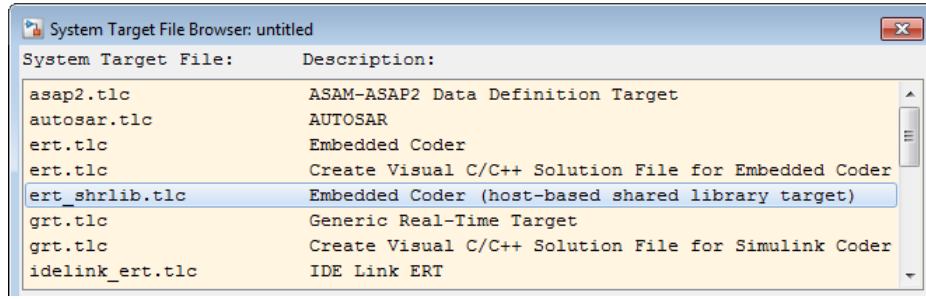
To generate and use a shared library:

1. Generate a shared library version of your model code
2. Create application code to load and use your shared library file

Generate Shared Library Version of Model Code

To generate a shared library version of your model code:

1. Open your model and configure it to use the `ert_shrlib.tlc` system target file.



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

2. Build the model.
3. After the build completes, examine the generated code in the model subfolder and examine the `.dll`, `.so`, or `.dylib` file in your current folder.

Create Application Code to Use Shared Library

To illustrate how application code can load a shared library file and access its functions and data, MathWorks provides the model `rtwdemo_shrlib`.

Note: Change directory to a writable working folder before running the `rtwdemo_shrlib` script.

In the model, click the blue button to run a script. The script:

1. Builds a shared library file from the model (for example, `rtwdemo_shrlib_win64.dll` on 64-bit Windows).
2. Compiles and links an example application, `rtwdemo_shrlib_app`, that loads and uses the shared library file.
3. Executes the example application.

Tip: Explicit linking is preferred for portability. But, on Windows systems, the `ert_shrlib` system target file generates and retains the `.lib` file to support implicit linking.

To use implicit linking, the generated *`model`*`.h` file needs a small modification for you to use it with t with the generated `ert_main.c`. For example, if you are using Visual C++, declare `__declspec(dllimport)` in front of data to be imported implicitly from the shared library file.

The model uses the following example application files, which are located in the folder *`matlabroot`*`/toolbox/rtw/rtwdemos/shrlib_demo` (open).

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 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 model window. Additionally, running the script places the relevant source and generated code files in your current folder. The files can be used as templates for writing application code for your own ERT shared library files.

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

Example Application Header File

The example application header file `rtwdemo_shrlib_app.h` contains type declarations for the 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 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 Script

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

Note: To run the `run_rtwdemo_shrlib_app` script without first opening the `rtwdemo_shrlib` model, change directory to a writable working folder and issue the following MATLAB command:

```
addpath(fullfile(matlabroot,'toolbox','rtw','rtwdemos','shrlib_demo'))
```

Shared Library Limitations

The following limitations apply to building shared libraries:

- Code generation for the `ert_shrlib.tlc` system target file exports the following as data:
  - Variables and signals of type `ExportedGlobal`
  - Real-time model structure (*`model`*`_M`)
- Code generation for the `ert_shrlib.tlc` system target file supports the C language only (not C++). When you select `ert_shrlib.tlc`, language selection is greyed out on the **Code Generation** pane of the Configuration Parameters dialog box.
- To reconstruct a model simulation using a generated 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 so that you can compare the simulation and integration results. Additional simulation considerations apply if generating a shared library from a model that enables parameters **Support: continuous time** and **Single output/update function**. For more information, see "Single output/update function" (Simulink Coder) dependencies.

More About

- "Design Models for Generated Embedded Code Deployment" on page 1-2

- "Select a System Target File" on page 30-2
- "Model Protection"

Real-Time Systems in Simulink Coder

Deploy Algorithm Model for Real-Time Rapid Prototyping

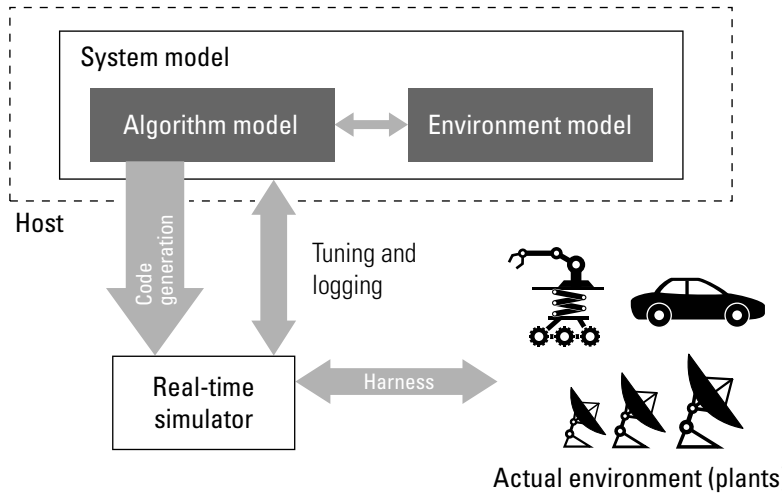
Use the code generator to deploy algorithm models for real-time rapid prototyping.

In this section...

About Real-Time Rapid Prototyping

Real-time rapid prototyping requires the use of a real-time simulator, potentially connected to system hardware (for example, physical plant or vehicle) being controlled. You generate, deploy, and tune code as it runs on the real-time simulator or embedded microprocessor. This design step is crucial for verifying whether a component can adequately control the system, and allows you to assess, interact with, and optimize code.

The following figure shows a typical approach for real-time rapid prototyping.



Goals of Real-Time Rapid Prototyping

Assuming that you have documented functional requirements, refined concept models, system hardware for the physical plant or vehicle being controlled, and access to target

products you intend to use (for example, for example, the Simulink Real-Time or Simulink Desktop Real-Time product), you can use real-time prototyping to:

- Refine component and environment model designs by rapidly iterating between algorithm design and prototyping
- Validate whether a component can adequately control the physical system in real time
- Evaluate system performance before laying out hardware, coding production software, or committing to a fixed design
- Test hardware

Refine Code With Real-Time Rapid Prototyping

To perform real-time rapid prototyping:

1. Create or acquire a real-time system that runs in real time on rapid prototyping hardware. The Simulink Real-Time product facilitates real-time rapid prototyping. This product provides a real-time operating system that makes PCs run in real time. It also provides device driver blocks for numerous hardware I/O cards. You can then create a rapid prototyping system using inexpensive commercial-off-the-shelf (COTS) hardware. In addition, third-party vendors offer products based on the Simulink Real-Time product or other code generation technology that you can integrate into a development environment.
2. Use provided system target files to generate code that you can deploy onto a real-time simulator. See the following information.

Engineering Tasks	Related Product Information	Examples
Generate code for real-time rapid prototyping	"Compare System Target File Support" (Simulink Coder) "Event-Based Scheduling" (Simulink Coder) Embedded Coder "Support for Standards and Guidelines" on page 12-2	`rtwdemo_counter` `rtwdemo_counter_msvc`

Engineering Tasks	Related Product Information	Examples
Generate code for rapid prototyping in hard real time, using PCs	Simulink Real-Time "Simulink Real-Time Options Pane" (Simulink Real-Time)	`help xpcdemos`
Generate code for rapid prototyping in soft real time, using PCs	Simulink Desktop Real-Time "Simulink Desktop Real-Time Pane" (Simulink Desktop Real-Time)	`sldrtex_vdp` (and others)

3 Monitor signals, tune parameters, and log data.

More About

- "Access Signal, State, and Parameter Data During Execution" on page 19-3
- "Rapid Control Prototyping Process" (Simulink Real-Time)

Deploy Environment Model for Real-Time Hardware-In-the-Loop (HIL) Simulation

In this section...

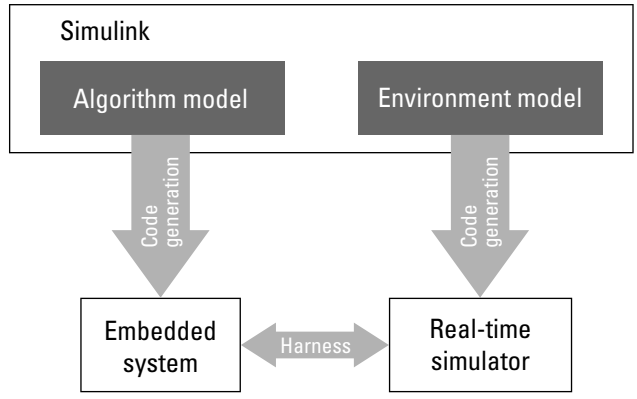
About Hardware-In-the-Loop Simulation

Hardware-in-the-loop (HIL) simulation tests and verifies an embedded system or control unit in the context of a software test platform. Examples of test platforms include real-time target systems and instruction set simulators (IISs). You use Simulink software to develop and verify a model that represents the test environment. Using the code generator, you produce, build, and download an executable program for the model to the HIL simulation platform. After you set up the environment, you can run the executable to validate the embedded system or control unit in real time.

During HIL simulation, you gradually replace parts of a system environment with hardware components as you refine and fabricate the components. HIL simulation offers an efficient design process that eliminates costly iterations of part fabrication.

The code that you build for the system simulator provides real-time system capabilities. For example, the code can include VxWorks from Wind River or another real-time operating system (RTOS).

The following figure shows a typical HIL setup.



The HIL platform available from MathWorks is the Simulink Real-Time product. Several third-party products are also available for use as HIL platforms. The Simulink Real-Time product offers hard real-time performance for PCs with Intel or AMD® 32-bit processors functioning as your real-time target. The Simulink Real-Time product enables you to add I/O interface blocks to your models and automatically generate code with code generation technology. The Simulink Real-Time product can download the code to a second PC running the Simulink Real-Time real-time kernel. System integrator solutions that are based on Simulink Real-Time are also available.

Set Up and Run HIL Simulations

To set up and run HIL simulations iterate through the following steps:

1. Develop a model that represents the environment or system under development.

   For more information, see "Compare System Target File Support" (Simulink Coder).
2. Generate an executable for the environment model.
3. Download the executable for the environment model to the HIL simulation platform.
4. Replace software representing a system component with corresponding hardware.
5. Test the hardware in the context of the HIL system.
6. Repeat steps 4 and 5 until you can simulate the system after including components that require testing.

More About

- "Access Signal, State, and Parameter Data During Execution" on page 19-3
- "Hardware-In-The-Loop Simulation Process" (Simulink Real-Time)

Real-Time and Embedded Systems in Embedded Coder

Deploy Generated Standalone Executable Programs To Target Hardware

By default, the Embedded Coder software generates *standalone* executable programs that do not require an external real-time executive or operating system. A standalone program requires minimal modification to be adapted to the target hardware. The standalone program architecture supports execution of models with either single or multiple sample rates.

In this section...

Generate a Standalone Program

To generate a standalone program:

1. In the **Custom templates** section of the **Code Generation > Templates** pane of the Configuration Parameters dialog box, select the **Generate an example main program** option (which is on by default). This enables the **Target operating system** menu.
2. From the **Target operating system** menu, select `BareBoardExample` (the default selection).
3. Generate the code.

Different code is generated for multirate models depending on the following factors:

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

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

Standalone Program Components

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

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

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

Main Program

Overview of Operation

The following pseudocode shows the execution of a 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 main 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 the main program module to implement your harness program.

1. Call *`model`*`_initialize`.
2. Initialize target-specific data structures and hardware, such as ADCs or DACs.
3. Install `rt_OneStep` as a timer ISR.
4. Initialize timer hardware.
5. Enable timer interrupts and start the timer.

   Note `rtModel` is not in a valid state until *`model`*`_initialize` has been called. Servicing of timer interrupts should not begin until *`model`*`_initialize` has been called.
6. Optionally, insert background task calls in the main loop.
7. On termination of the main loop (if applicable):

   - Disable timer interrupts.
   - Perform target-specific cleanup such as zeroing DACs.
   - Detect and handle errors. Note that even if your program is designed to run indefinitely, you may need to handle severe error conditions, such as timer interrupt overruns.

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

rt_OneStep and Scheduling Considerations

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. A model in which the sample times and step size do not meet these conditions is termed multirate.
- Your model's solver mode (`SingleTasking` versus `MultiTasking`)

Permitted Solver Modes for Embedded Real-Time System Target Files 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 Embedded Real-Time System Target Files

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

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

Single-Rate Single-Tasking Operation

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

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

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

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

```
void model_step(void)
```

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

The overrun flag is cleared only upon successful return from *`model`*`_step`. Therefore, if `rt_OneStep` is reinterrupted before completing *`model`*`_step`, the 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, code generation uses a prioritized, preemptive multitasking scheme to execute the different sample rates in your model.

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

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

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

  For N=1:NumTasks-1   -- iterate over sub-rate tasks
    If (sub-rate task N is scheduled)
    Check for sub-rate interrupt overrun
      Model_StepN()    -- run sub-rate time step code
    EndIf
  EndFor
}
```

Task Identifiers

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

Prioritization of Base-Rate and Subrate Tasks

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

The slower tasks, running at multiples of the base rate, are called *subrate* tasks.

Rate Grouping and Rate-Specific model_step Functions

In a single-rate model, the block output computations are performed within a single function, *`model`*`_step`. For multirate, multitasking models, the code generator tries to use a different strategy. This strategy is called *rate grouping*. Rate grouping generates separate *`model`*`_step` functions for the base rate task and each subrate task in the model. The function naming convention for these functions is

`model``_step`*`N`*

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 the blocks sharing `tid` *`N`*; in other words, the 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` maintains scheduling counters and *event flags* for each subrate task. The counters are implemented as `taskCounter` arrays indexed on `tid`. The event flags are implemented as arrays indexed on `tid`.

The scheduling counters and task flags for sub-rates are maintained by `rt_OneStep`. The scheduling counters are basically clock rate dividers that count up the sample period

associated with each sub-rate task. A pair of tasks that exchanges data maintains an interaction flag at the faster rate. Task interaction flags indicate that both fast and slow tasks are scheduled to run.

The event flags indicate whether or not a given task is scheduled for execution. `rt_OneStep` maintains the event flags based on a task counter that is maintained by code in the main program module for the model. When a counter indicates that a task's sample period has elapsed, the main code sets the event flag for that task.

On each invocation, `rt_OneStep` updates its scheduling data structures and steps the base-rate task (`rt_OneStep` 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. The 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. Therefore, `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 49-17 for information about how to adapt your code for rate grouping compatibility. This adaptation lets your multirate, multitasking models generate more efficient code.

Multirate Single-Tasking Operation

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

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

```
void model_step(void)
```

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

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

Guidelines for Modifying rt_OneStep

`rt_OneStep` does not require extensive modification. The only required modification is to reenable interrupts after the overrun flags and error conditions have been checked. If applicable, you should also

- Save and restore your FPU context on entry and exit to `rt_OneStep`.
- Set model inputs associated with the base rate before calling *`model`*`_step0`.
- Get model outputs associated with the base rate after calling *`model`*`_step0`.

Note: If you modify `rt_OneStep` to read a value from a continuous output port after each base-rate model step, see the relevant cautionary guideline below.

- In a multirate, multitasking model, set model inputs associated with subrates before calling *`model`*`_step`*`N`* in the subrate loop.
- In a multirate, multitasking model, get model outputs associated with subrates after calling *`model`*`_step`*`N`* in the subrate loop.

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

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

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

Also observe the following cautionary guidelines:

- You should not modify the way in which the counters, event flags, or other timing data structures are set in `rt_OneStep`, or in functions called from `rt_OneStep`. The `rt_OneStep` timing data structures (including `rtModel`) and logic are critical to the operation of the generated program.
- If you have customized the main program module to read model outputs after each base-rate model step, be aware that selecting model options **Support: continuous time** and **Single output/update function** together may cause output values read from `main` for a continuous output port to differ slightly from the corresponding output values in the model's logged data. This is because, while logged data is a snapshot of output at major time steps, output read from `main` after the base-rate model step potentially reflects intervening minor time steps. To eliminate the discrepancy, either separate the generated output and update functions (clear the **Single output/update function** option) or place a Zero-Order Hold block before the continuous output port.
- It is possible to observe a mismatch between results from simulation and logged MAT file results from generated code if you do not set model inputs before each time you call the model step function. In the generated example main program, the following comments show the locations for setting the inputs and stepping the model with your code:

  ```
  /* Set model inputs here */
  /* Step the model */
  ```

  If your model applies signal reuse and you are using `MatFileLogging` for comparing results from simulation against generated code, modify `rt_OneStep` to write model inputs in every time step as directed by these comments. Alternatively, you could “Choose a SIL or PIL Approach” on page 64-11 for verification.

Static Main Program Module

Overview

In most cases, the easiest strategy for deploying generated code is to use the **Generate an example main program option** to generate the `ert_main.c` or `.cpp` module (see "Generate a Standalone Program" on page 49-2).

However, if you turn the **Generate an example main program** option off, you can use a static main module as an example or template for developing your embedded applications. Static main modules provided by MathWorks include:

- *matlabroot*`/rtw/c/src/common/rt_main.c` — Supports `Nonreusable function` code interface packaging.
- *matlabroot*`/rtw/c/src/common/rt_malloc_main.c` — Supports `Reusable function` code interface packaging. The model option **Use dynamic memory allocation for model initialization** must be on and model parameter **Pass root-level I/O as** must be set to `Part of model data structure`.
- *matlabroot*`/rtw/c/src/common/rt_cppclass_main.cpp` — Supports `C++ class` code interface packaging.

The static main 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 depend upon a static `ert_main.c` (developed in releases before R2012b), `rt_main.c`, `rt_malloc_main.c`, or `rt_cppclass_main.cpp`, you may need to continue using a static main program module.

When developing applications using a static main module, you should copy the module to your working folder and rename it before making modifications. For example, you could rename `rt_main.c` to *`model`*`_rt_main.c`. Also, you must modify the template makefile or toolchain settings such that the build process creates a corresponding object file, such as *`model`*`_rt_main.obj` (on UNIX, *`model`*`_rt_main.o`), in the build folder.

The static main module 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 main module as they are in the automatically generated version described in "Deploy Generated Standalone Executable Programs To

Target Hardware" on page 49-2. For multirate, multitasking models, however, the static and generated code are 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 main module and disallow use of the **Generate an example main program** option. This is done because target-specific modifications have been added to the static main module, and these modifications would not be preserved if the main program were regenerated.

Your static main module may or may not use rate grouping compatible *model*_step*N* functions. If your main module is based on the static `rt_main.c`, `rt_malloc_main.c`, or `rt_cppclass_main.cpp` module, it does not use rate-specific *model*_step*N* function calls. It 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 *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.

Modify the Static Main Program

As with the generated `ert_main.c` or `.cpp`, you should make a few modifications to the main loop and `rt_OneStep`. See "Guidelines for Modifying the Main Program" on page 49-4 and "Guidelines for Modifying rt_OneStep" on page 49-9.

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 applicable, follow comments in the code regarding where to add code for reading/writing model I/O and saving/restoring FPU context.

Note: If you modify `rt_main.c`, `rt_malloc_main.c`, or `rt_cppclass_main.cpp` to read a value from a continuous output port after each base-rate model step, see the relevant cautionary guideline in "Guidelines for Modifying rt_OneStep" on page 49-9.

- When the **Generate an example main program** option is off, `rtmodel.h` is generated 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 `rtmodel.h`.

  Alternatively, you can suppress generation of `rtmodel.h`, and include *`model`*`.h` directly in your main module. To suppress generation of `rtmodel.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 `rt_main.c`, `rt_malloc_main.c`, or `rt_cppclass_main.cpp`:

  - The `#if TERMFCN...` compile-time error check
  - The call to `MODEL_TERMINATE`

- For `rt_main.c` (not applicable to `rt_cppclass_main.cpp`): 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 `rt_main.c`:

  - Replace calls to `MODEL_STEP` with calls to `MODEL_OUTPUT` and `MODEL_UPDATE`.
  - Remove the `#if ONESTEPFCN...` error check.

- The static `rt_main.c` module does not support `Reusable function` code interface packaging. The following error check raises a compile-time error if `Reusable function` code interface packaging is used illegally.

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

Modify Static Main to Allocate and Access Model Instance Data

If you are using a static main program module, and your model is configured for `Reusable function` code interface packaging, but the model option **Use dynamic memory allocation for model initialization** is not selected, model instance data must be allocated either statically or dynamically by the calling main code. Pointers to the individual model data structures (such as Block IO, DWork, and Parameters) must be set up in the top-level real-time model data structure.

To support main modifications, the build process generates a subset of the following real-time model (RTM) macros, based on the data requirements of your model, into *`model`*`.h`.

RTM Macro Syntax	Description
`rtmGetBlockIO(rtm)`	Get the block I/O data structure
`rtmSetBlockIO(rtm,val)`	Set the block I/O data structure
`rtmGetContStates(rtm)`	Get the continuous states data structure
`rtmSetContStates(rtm,val)`	Set the continuous states data structure
`rtmGetDefaultParam(rtm)`	Get the default parameters data structure
`rtmSetDefaultParam(rtm,val)`	Set the default parameters data structure
`rtmGetPrevZCSigState(rtm)`	Get the previous zero-crossing signal state data structure
`rtmSetPrevZCSigState(rtm,val)`	Set the previous zero-crossing signal state data structure
`rtmGetRootDWork(rtm)`	Get the DWork data structure
`rtmSetRootDWork(rtm,val)`	Set the DWork data structure
`rtmGetU(rtm)`	Get the root inputs data structure (when root inputs are passed as part of the model data structure)
`rtmSetU(rtm,val)`	Set the root inputs data structure (when root inputs are passed as part of the model data structure)
`rtmGetY(rtm)`	Get the root outputs data structure (when root outputs are passed as part of the model data structure)
`rtmSetY(rtm,val)`	Set the root outputs data structure (when root outputs are passed as part of the model data structure)

Use these macros in your static main program to access individual model data structures within the RTM data structure. For example, suppose that the example model `rtwdemo_reusable` is configured with `Reusable function` code interface packaging, **Use dynamic memory allocation for model initialization** cleared, **Pass root-level I/O as** set to `Individual arguments`, and **Optimization** pane option **Remove root level I/O zero initialization** cleared. Building the model generates the following model data structures and model entry-points into `rtwdemo_reusable.h`:

```
/* Block states (auto storage) for system '<Root>' */
typedef struct {
  real_T Delay_DSTATE;                 /* '<Root>/Delay' */
} D_Work;
```

```
/* Parameters (auto storage) */
struct Parameters_ {
  real_T k1;                           /* Variable: k1
                                        * Referenced by: '<Root>/Gain'
                                        */
};

/* Model entry point functions */
extern void rtwdemo_reusable_initialize(RT_MODEL *const rtM, real_T *rtU_In1,
  real_T *rtU_In2, real_T *rtY_Out1);
extern void rtwdemo_reusable_step(RT_MODEL *const rtM, real_T rtU_In1, real_T
  rtU_In2, real_T *rtY_Out1);
```

Additionally, if **Generate an example main program** is not selected for the model, `rtwdemo_reusable.h` contains definitions for the RTM macros `rtmGetDefaultParam`, `rtmsetDefaultParam`, `rtmGetRootDWork`, and `rtmSetRootDWork`.

Also, for reference, the generated `rtmodel.h` file contains an example parameter definition with initial values (non-executing code):

```
#if 0

/* Example parameter data definition with initial values */
static Parameters rtP = {
  2.0                                  /* Variable: k1
                                        * Referenced by: '<Root>/Gain'
                                        */
};                                     /* Modifiable parameters */

#endif
```

In the definitions section of your static main file, you could use the following code to statically allocate the real-time model data structures and arguments for the `rtwdemo_reusable` model:

```
static RT_MODEL rtM_;
static RT_MODEL *const rtM = &rtM_;    /* Real-time model */
static Parameters rtP = {
  2.0                                  /* Variable: k1
                                        * Referenced by: '<Root>/Gain'
                                        */
};                                     /* Modifiable parameters */

static D_Work rtDWork;                 /* Observable states */

/* '<Root>/In1' */
static real_T rtU_In1;

/* '<Root>/In2' */
static real_T rtU_In2;

/* '<Root>/Out1' */
```

```
static real_T rtY_Out1;
```

In the body of your main function, you could use the following RTM macro calls to set up the model parameters and DWork data in the real-time model data structure:

```
int_T main(int_T argc, const char *argv[])
{
...
/* Pack model data into RTM */

rtmSetDefaultParam(rtM, &rtP);
rtmSetRootDWork(rtM, &rtDWork);

/* Initialize model */
rtwdemo_reusable_initialize(rtM, &rtU_In1, &rtU_In2, &rtY_Out1);
...
}
```

Follow a similar approach to set up multiple instances of model data, where the real-time model data structure for each instance has its own data. In particular, the parameter structure (`rtP`) should be initialized, for each instance, to the desired values, either statically as part of the `rtP` data definition or at run time.

Rate Grouping Compliance and Compatibility Issues

Main Program Compatibility

When the **Generate an example main program** option is off, code generation produces 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 49-12 for details.

Make Your S-Functions Rate Grouping Compliant

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

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

- Reduced code efficiency.
- Warning messages issued at compile time. Such warnings are caused when dead code references temporary variables before initialization. Since the dead code does not run, 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
```

More About

- "Design Models for Generated Embedded Code Deployment" on page 1-2

Deploy Generated Component Software to Application Target Platforms

The code generator supports integration of generated code with operating systems and processors. For details, see “Embedded Coder Supported Hardware” on page 68-2.

Interface to an Example Real-Time Operating System (VxWorks®)

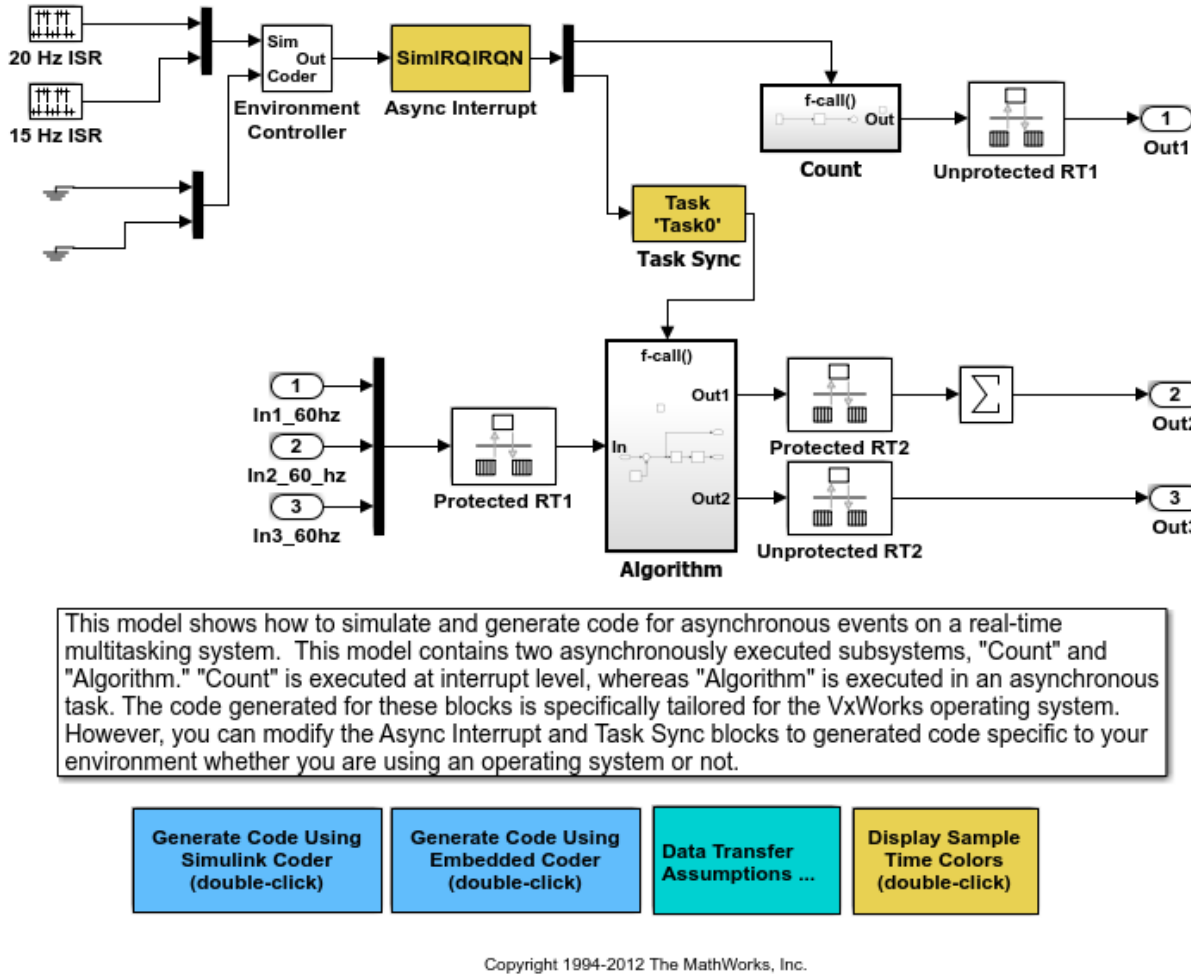
This example shows how to simulate and generate code for asynchronous events on an example RTOS (VxWorks).

The operating system integration techniques that are demonstrated in this example use one or more blocks the blocks in the `vxlib1` library. These blocks provide starting point examples to help you develop custom blocks for your target environment.

Example Model

Open the `rtwdemo_vxworks` model.

```
model = 'rtwdemo_vxworks';
open_system(model);
%
```



Model Description

The example model contains two asynchronously executed subsystems, `Count` and `Algorithm`. `Count` executes at interrupt level. `Algorithm` executes in an asynchronous task. The generated code for these blocks is tailored for the VxWorks® operating system. However, you can modify the `Async Interrupt` and `Task Sync` blocks to generate code for your run-time environment whether you are using an operating system or not.

Related Information

- Async Interrupt (Simulink Coder)
- Task Sync (Simulink Coder)
- “Generate Interrupt Service Routines” (Simulink Coder)
- “Timers in Asynchronous Tasks” (Simulink Coder)
- “Create a Customized Asynchronous Library” (Simulink Coder)
- “Import Asynchronous Event Data for Simulation” (Simulink Coder)
- “Load Data to Root-Level Input Ports” (Simulink)
- “Asynchronous Events” (Simulink Coder)
- “Rate Transitions and Asynchronous Blocks” (Simulink Coder)
- “Asynchronous Support Limitations” (Simulink Coder)

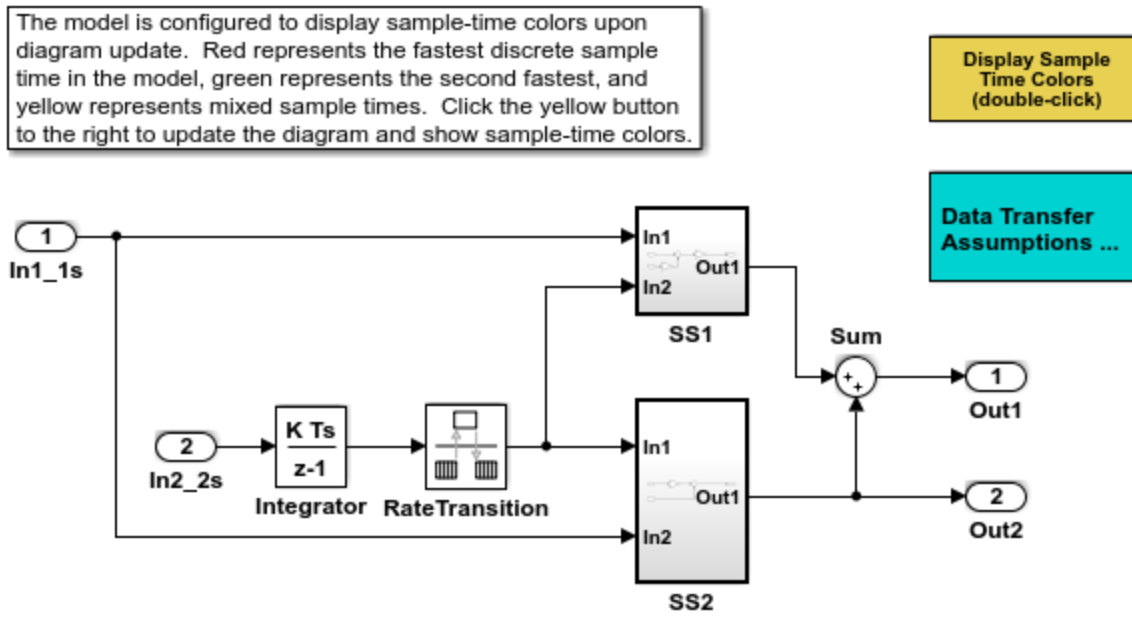
Multirate Modeling in Multitasking Mode (VxWorks® OS)

This example generates code for a multirate discrete-time model configured for a multitasking operating system target (VxWorks®). The model contains two sample times. Inport block 1 and Inport block 2 specify 1-second and 2-second sample times, respectively, which are enforced by the **Periodic sample time constraint** solver configuration parameter setting. The solver is set for multitasking operation, which means a Rate Transition block is required to ensure that data integrity is enforced when the 1-second task preempts the 2-second task. Simulink® and the code generator enforce proper rate transitions. This model specifies an explicit Rate Transition block. Alternatively, you can instruct Simulink® to insert this block for you by setting the **Automatically handle data transfers between tasks** solver configuration parameter.

The model is configured to display sample-time colors upon diagram update. Red represents the fastest discrete sample time in the model, green represents the second fastest, and yellow represents mixed sample times. Click the **Display Sample Time Colors** button to update the diagram and show sample-time colors.

Example Model

```
model = 'rtwdemo_mrmtos';
open_system(model);
```



This model shows the code generated for a multirate discrete-time model configured for a multitasking operating system target (VxWorks). The model contains two sample times. Inport block 1 and Inport block 2 specify 1-second and 2-second sample times, respectively, which are enforced by the "Periodic sample time constraint" option on the Solver configuration page. The solver is set for multitasking operation, which means a rate transition block is required to ensure that data integrity is enforced when the 1-second task preempts the 2-second task. Proper rate transitions are always enforced by Simulink and Simulink Coder. This model specifies an explicit rate transition block. Alternatively, this block could be automatically inserted by Simulink using the "Automatically handle data transfers between tasks" option on the Solver configuration page.

This example uses a Embedded Coder feature to generate an example VxWorks main (i.e., "Target operating system" on the Template configuration page.) Simulink Coder also supports VxWorks via the Tornado target.

Generate Code Using Embedded Coder (double-click)

More About

View Solver Configuration (double-click)

"Design Models for Generated Embedded Code Deployment" on page 1-2

Export Code Generated from Model to External Application in Embedded Coder

Control Generation of Function Prototypes

The Embedded Coder software provides a **Configure Model Functions** button, located on the **Code Generation** > **Interface** pane of the Configuration Parameters dialog box, that allows you to control the model function prototypes that are generated for ERT-based models.

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

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

The function prototype of an ERT-based model's generated *`model`*`_initialize` function resembles the following:

```
void model_initialize(void);
```

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

The **Configure Model Functions** button on the **Interface** pane provides you flexible control over the model function prototypes that are generated for your model. Clicking **Configure Model Functions** launches a Model Interface dialog box. Based on the **Function specification** value you specify for your model function (supported values include `Default model initialize and step functions` and `Model specific C prototypes`), you can preview and modify the function prototypes. Once you validate and apply your changes, you can generate code based on your function prototype modifications.

For more information about using the **Configure Model Functions** button and the Model Interface dialog box, see “Sample Procedure for Configuring Function Prototypes” on page 26-11 and the example model `rtwdemo_fcnprotoctrl`, which is preconfigured to demonstrate function prototype control.

Alternatively, you can use function prototype control functions to programmatically control model function prototypes. For more information, see “Configure Function Prototypes Programmatically” on page 26-16“Configure Function Prototypes Programmatically” on page 26-16.

You can also control model function prototypes for nonvirtual subsystems, if you generate subsystem code using right-click build. To launch the **Model Interface for subsystem** dialog box, use the `RTW.configSubsystemBuild` function.

Right-click building the subsystem generates the step and initialization functions according to the customizations you make. For more information, see “Configure Function Prototypes for Nonvirtual Subsystems” on page 26-9.

For limitations that apply, see “Function Prototype Control Limitations” on page 26-21.

More About

- “Design Models for Generated Embedded Code Deployment” on page 1-2
- “Entry-Point Functions and Scheduling” on page 25-2
- “Generate Component Source Code for Export to External Code Base” on page 39-51
- “Export-Function Models” (Simulink)

Control Generation of C++ Class Interfaces

Using the **Code interface packaging** (Simulink Coder) option `C++ class`, on the **Code Generation** > **Interface** pane of the Configuration Parameters dialog box, you can generate a C++ class interface to model code. The generated interface encapsulates required model data into C++ class attributes and model entry point functions into C++ class methods. The benefits of C++ class encapsulation include:

- Greater control over access to model data
- Ability to multiply instantiate model classes
- Easier integration of model code into C++ programming environments

C++ class encapsulation also works for right-click builds of nonvirtual subsystems. (For information on requirements that apply, see "Configure C++ Class Interfaces for Nonvirtual Subsystems" on page 26-44.)

The general procedure for generating C++ class interfaces to model code is as follows:

1. Configure your model to use an `ert.tlc` system target file provided by MathWorks.
2. Select the `C++` language for your model.
3. Select `C++ class` code interface packaging for your model.
4. Optionally, configure related C++ class interface settings for your model code, using either a graphical user interface (GUI) or application programming interface (API).
5. Generate model code and examine the results.

To get started with an example, see "Simple Use of C++ Class Control" on page 26-24. For more details about configuring C++ class interfaces for your model code, see "Customize C++ Class Interfaces Using Graphical Interfaces" on page 26-31 and "Customize C++ Class Interfaces Programmatically" on page 26-45. For limitations that apply, see "C++ Class Interface Control Limitations" on page 26-50.

Note: For an example of C++ class code generation, see the example model `rtwdemo_cppclass`.

More About

- "Design Models for Generated Embedded Code Deployment" on page 1-2

- "Entry-Point Functions and Scheduling" on page 25-2
- "Generate Component Source Code for Export to External Code Base" on page 39-51
- "Export-Function Models" (Simulink)

Code Replacement Customization for Simulink Models in Embedded Coder

What Is Code Replacement Customization?

Customize how and when the code generator replaces C/C++ code that it generates by default for functions and operators by developing a custom code replacement library. You can develop libraries interactively with the Code Replacement Tool or programmatically.

- Develop libraries tailored to specific application requirements
- Add identifiers to the list of reserved keywords the code generator considers during code replacement
- Customize the code generator's match and replacement process for functions

To get started, "Quick Start Library Development" on page 51-28.

Code Replacement Match and Replacement Process

When the code generator encounters a call site for a function or operator, it:

1. Creates and partially populates a code replacement entry object with the function or operator name or key and conceptual arguments.
2. Uses the entry object to query the configured code replacement library for a conceptual representation match. The code generator searches the tables in a code replacement library for a match in the order that the tables appear in the library. When searching for a match, the code generator takes into account:
   - Conceptual name or key
   - Arguments, including quantity, type, type qualifiers, and complexity
   - Algorithm (computation method)
   - Fixed-point saturation and rounding modes
   - Priority
3. When a match exists, the code generator returns a code replacement object, fully populated with the conceptual representation, implementation representation, and priority. If the code generator finds multiple matches within a table, the entry priority determines the match. The priority can range from 0 to 100. The highest priority is 0. The code generator uses a higher-priority entry over a similar entry with a lower priority.
4. Uses the C or C++ replacement function prototype in the code replacement object to generate code.

Code Replacement Customization Limitations

- Code replacement verification — It is possible that code replacement behaves differently than you expect. For example, data types that you observe in code generator input might not match what the code generator uses as intermediate data types during an operation. Verify code replacements by examining generated code. See "Verify Code Replacements" on page 51-76.
- Tokens in file paths—You can include tokens in file paths when specifying build information for a code replacement entry by using the programming interface only. The ability to include tokens is not available from the Code Replacement Tool. See "Specify Build Information for Replacement Code" on page 51-59.
- Addition and subtraction operation replacements—See"Addition and Subtraction Operator Code Replacement" on page 51-170 for relevant limitations.
- Data alignment—
  - Not supported for
    - Arguments associated with a built-in custom storage class with `DataScope` set to `Exported` or the imported built-in custom storage class `GetSet`
    - Software-in-the-loop (SIL)
    - Processor-in-the-loop (PIL)
    - Model reference parameters
    - Exported functions in Stateflow charts
    - Replaced functions that are generated with C function prototype control or C++ class I/O arguments step method and that use root-level I/O variables
    - Replaced functions that are generated with the AUTOSAR system target file and that use root-level I/O or AUTOSAR inter-runnable access functions
  - If the following conditions exist, the code generator includes data alignment directives for root-level I/O variables in the `ert_main.c` or `ert_main.cpp` file it produces:
    - Compiler supports global variable alignment
    - Generate an example main program (select **Configuration Parameters** > **All Parameters** > **Generate an example main program**)
    - Generate a reusable function interface for the model (set **Configuration Parameters** > **Code Generation** > **Interface** > **Code interface packaging** to `Reusable function`)

  - Function uses root-level I/O variables that are passed in as individual arguments (set **Configuration Parameters** > **Code Generation** > **Interface** > **Pass root-level I/O as**to `Individual arguments`)
  - Replaced function uses a root-level I/O variable
  - Replaced function imposes alignment requirements

  If you discard the generated example main program, align used root-level I/O variables correctly.

  If you choose not to generate an example main program in this case, the code generator does not replace the function.
- If a replacement imposes alignment requirements on the shared utility interface arguments, the code generator does not honor data alignment. Under these conditions, replacement does not occur. Replacement is allowed if the registered data alignment type specification supports alignment of local variables, and the replacement involves only local variables.
- For `Simulink.Bus`:
  - If user registered alignment specifications do not support structure field alignment, aligning `Simulink.Bus` objects is not supported unless the `Simulink.Bus` is imported.
  - When aligning a `Simulink.Bus` data object, the elements in the bus object are aligned on the same boundary. The boundary is the lowest common multiple of the alignment requirements for each individual bus element.
- When you specify alignment for functions that occur in a model reference hierarchy, and multiple models in the hierarchy operate on the same function data, the bottommost model dictates alignment for the rest of the hierarchy. If the alignment requirement for a function in a model higher in the hierarchy cannot be honored due to the alignment set by a model lower in the hierarchy, the replacement in the higher model does not occur. In some cases, an error message is generated. To work around this issue, if the shared data is represented by a bus or signal object, manually set the alignment property on the shared data by setting the alignment property of the `Simulink.Bus` or `Simulink.Signal` object.
- It is your responsibility to honor the `Alignment` property setting for custom storage classes that you create.

See “Data Alignment for Code Replacement” on page 51-133.

- `coder.replace` function — See `coder.replace` for relevant limitations.

More About

- "Code You Can Replace From Simulink Models" on page 51-7
- "Develop a Code Replacement Library" on page 51-27
- "Quick Start Library Development" on page 51-28
- "What Is Code Replacement?" on page 37-2

Code You Can Replace From Simulink Models

Code that the code generator replaces depends on the code replacement library (CRL) that you use. By default, the code generator does not apply a code replacement library. Your choice of libraries is dependent on product licensing and whether you have access to custom libraries.

For information on how to explore functions and operators that a code replacement library supports, see "Choose a Code Replacement Library" on page 38-9 license and want to develop a custom code replacement library, see Code Replacement Customization.

In this section...

Math Functions – Simulink Support

When generating C/C++ code from a Simulink model, depending on code replacement libraries available in your development environment, you can configure the code generator to replace instances of the following math functions with application-specific implementations.

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
`abs`[1]	Integer Floating point Fixed point	Scalar Vector Matrix	Real
`acos`	Floating point	Scalar	Real Complex input/complex output Real input/complex output
`acosd`[2]	Floating point	Scalar Vector	Real Complex

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
		Matrix	
`acosh`	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
`acot`[2]	Floating point	Scalar Vector Matrix	Real Complex
`acotd`[2]	Floating point	Scalar Vector Matrix	Real Complex
`acoth`[2]	Floating point	Scalar Vector Matrix	Real Complex
`acsc`[2]	Floating point	Scalar Vector Matrix	Real Complex
`acscd`[2]	Floating point	Scalar Vector Matrix	Real Complex
`acsch`[2]	Floating point	Scalar Vector Matrix	Real Complex
`asec`[2]	Floating point	Scalar Vector Matrix	Real Complex
`asecd`[2]	Floating point	Scalar Vector Matrix	Real Complex
`asech`[2]	Floating point	Scalar Vector Matrix	Real Complex

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
`asin`	Floating point	Scalar	Real Complex input/complex output Real input/complex output
`asind`[2]	Floating point	Scalar Vector Matrix	Real Complex
`asinh`	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
`atan`	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
`atan2`	Floating point	Scalar Vector Matrix	Real
`atan2d`[2]	Floating point	Scalar Vector Matrix	Real
`atand`[2]	Floating point	Scalar Vector Matrix	Real Complex
`atanh`	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
`ceil`	• Floating-point • Scalar	• Floating-point • Scalar	• Floating-point • Scalar
`cos`[3]	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
`cosd`[2]	Floating point	Scalar Vector Matrix	Real Complex

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
`cosh`	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
`cot`[2]	Floating point	Scalar Vector Matrix	Real Complex
`cotd`[2]	Floating point	Scalar Vector Matrix	Real Complex
`coth`[2]	Floating point	Scalar Vector Matrix	Real Complex
`csc`[2]	Floating point	Scalar Vector Matrix	Real Complex
`cscd`[2]	Floating point	Scalar Vector Matrix	Real Complex
`csch`[2]	Floating point	Scalar Vector Matrix	Real Complex
`exactrSqrt`	Integer Floating point	Scalar	Real
`exp`	Floating point	Scalar Vector Matrix	Real
`fix`	Floating point	Scalar	Real
`floor`	• Floating-point • Scalar	• Floating-point • Scalar	• Floating-point • Scalar
`fmod`[4]	Floating point	Scalar	Real
`frexp`	Floating point	Scalar	Real

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
`hypot`	Floating point	Scalar Vector Matrix	Real
`ldexp`	Floating point	Scalar	Real
`ln`	Floating point	Scalar	Real
`log`	Floating point	Scalar Vector Matrix	Real
`log10`	Floating point	Scalar Vector Matrix	Real
`log2`[2]	Floating point	Scalar Vector Matrix	Real Complex
`max`	Integer Floating point Fixed point	Scalar	Real
`min`	Integer Floating point Fixed point	Scalar	Real
`mod`	Integer Floating point	Scalar Vector Matrix	Real
`pow`	Floating point	Scalar Vector Matrix	Real
`rem`	Floating point	Scalar Vector Matrix	Real
`round`	Floating point	Scalar	Real

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
`rSqrt`	Integer Floating point	Scalar Vector Matrix	Real
`saturate`	Integer Floating point Fixed point	Scalar Vector Matrix	Real
`sec`[2]	Floating point	Scalar Vector Matrix	Real Complex
`secd`[2]	Floating point	Scalar Vector Matrix	Real Complex
`sech`[2]	Floating point	Scalar Vector Matrix	Real Complex
`sign`	Integer Floating point Fixed point	Scalar Vector Matrix	Real
`signPow`	Floating point	Scalar Vector Matrix	Real
`sin`[3]	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
`sincos`[3]	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
`sind`[2]	Floating point	Scalar Vector Matrix	Real Complex
`sinh`	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
`sqrt`	Integer Floating point Fixed point	Scalar Vector Matrix	Real
`tan`	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output
`tand`[2]	Floating point	Scalar Vector Matrix	Real Complex
`tanh`	Floating point	Scalar Vector Matrix	Real Complex input/complex output Real input/complex output

[1] Wrap on integer overflow only. Clear block parameter **Saturate on integer overflow**.

[2] Only when used with the MATLAB Function block.

[3] Supports the CORDIC approximation method.

[4] Stateflow support only.

Math Functions – Stateflow Support

When generating C/C++ code from Stateflow charts, depending on code replacement libraries available in your development environment, you can configure the code generator to replace instances of the following math functions with application-specific implementations.

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
`abs`[1]	Integer Floating point	Scalar	Real
`acos`[2]	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
			Real input/complex output
`acosd`[3]	Floating point	Scalar Vector Matrix	Real Complex
`acot`[3]	Floating point	Scalar Vector Matrix	Real Complex
`acotd`[3]	Floating point	Scalar Vector Matrix	Real Complex
`acoth`[3,5]	Floating point	Scalar Vector Matrix	Real Complex
`acsc`[3]	Floating point	Scalar Vector Matrix	Real Complex
`acscd`[3]	Floating point	Scalar Vector Matrix	Real Complex
`acsch`[3]	Floating point	Scalar Vector Matrix	Real Complex
`asec`[3]	Floating point	Scalar Vector Matrix	Real Complex
`asecd`[3]	Floating point	Scalar Vector Matrix	Real Complex
`asech`[3]	Floating point	Scalar Vector Matrix	Real Complex

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
`asin`[2]	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
`asind`[3]	Floating point	Scalar Vector Matrix	Real Complex
`atan`[2]	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
`atan2`[2]	Floating point	Scalar Vector Matrix	Real
`atan2d`[3]	Floating point	Scalar Vector Matrix	Real
`atand`[3]	Floating point	Scalar Vector Matrix	Real Complex
`ceil`	• Floating-point • Scalar	• Floating-point • Scalar	• Floating-point • Scalar
`cos`[3]	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
`cosd`[3]	Floating point	Scalar Vector Matrix	Real Complex

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
`cosh`[2]	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
`cot`[3]	Floating point	Scalar Vector Matrix	Real Complex
`cotd`[3]	Floating point	Scalar Vector Matrix	Real Complex
`coth`[3]	Floating point	Scalar Vector Matrix	Real Complex
`csc`[3]	Floating point	Scalar Vector Matrix	Real Complex
`cscd`[3]	Floating point	Scalar Vector Matrix	Real Complex
`csch`[3]	Floating point	Scalar Vector Matrix	Real Complex
`exp`	Floating point	Scalar	Real
`floor`	• Floating-point • Scalar	• Floating-point • Scalar	• Floating-point • Scalar
`fmod`	Floating point	Scalar	Real
`hypot`[3]	Floating point	Scalar Vector Matrix	Real
`ldexp`	Floating point	Scalar	Real

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
`log`[2]	Floating point	Scalar Vector Matrix	Real Complex
`log10`[2]	Floating point	Scalar Vector Matrix	Real Complex
`log2`[3]	Floating point	Scalar Vector Matrix	Real Complex
`max`	Integer Floating point	Scalar	Real
`min`	Integer Floating point	Scalar	Real
`pow`	Floating point	Scalar	Real
`sec`[3]	Floating point	Scalar Vector Matrix	Real Complex
`secd`[3]	Floating point	Scalar Vector Matrix	Real Complex
`sech`[3]	Floating point	Scalar Vector Matrix	Real Complex
`sin`[2]	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
`sind`[3]	Floating point	Scalar Vector Matrix	Real Complex

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
`sinh`[2]	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
`sqrt`	Floating point	Scalar	Real
`tan`[2]	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
`tand`[3]	Floating point	Scalar Vector Matrix	Real Complex
`tanh`[2]	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output

[1] Wrap on integer overflow only.

[2] For models involving vectors or matrices, the code generator replaces only functions coded in the MATLAB action language.

[3] The code generator replaces only functions coded in the MATLAB action language.

Memory Functions

Depending on code replacement libraries available in your development environment, you can configure the code generator to replace instances of the following memory functions with application-specific implementations.

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
`memcmp`	Void pointer (`void*`)	Scalar	Real

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
		Vector Matrix	Complex
`memcpy`	Void pointer (`void*`)	Scalar Vector Matrix	Real Complex
`memset`	Void pointer (`void*`)	Scalar Vector Matrix	Real Complex
`memset2zero`	Void pointer (`void*`)	Scalar Vector Matrix	Real Complex

Some target processors provide optimized functions to set memory to zero. Use the code replacement library programming interface to replace the `memset2zero` function with more efficient target-specific functions.

Nonfinite Functions

Depending on code replacement libraries available in your development environment, you can configure the code generator to replace instances of the following nonfinite functions with application-specific implementations.

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
`getInf`	Floating point	Scalar	Real
`getMinusInf`	Floating point	Scalar	Real
`getNaN`	Floating point	Scalar	Real
`rtIsInf`	Floating point	Scalar	Real Complex
`rtIsNaN`	Floating point	Scalar	Real Complex

Mutex and Semaphore Functions

Mutex and semaphore functions control access to resources shared by multiple processes in multicore target environments. MathWorks provides code replacement libraries that support mutex and semaphore replacement for Rate Transition and Task Transition blocks on Windows, Linux, Mac, and VxWorks platforms.

Generated mutex and semaphore code typically consists of:

- In model initialization code, an initialization function call to create a mutex or semaphore to control entry to a critical section of code.
- In model step code:
  - Before code for a data transfer between tasks enters the critical section, mutex lock or semaphore wait function calls to reserve a critical section of code.
  - After code for a data transfer between tasks finishes executing the critical section, mutex unlock or semaphore post function calls to release the critical section of code.
- In model termination code, an optional destroy function call to explicitly delete the mutex or semaphore.

Depending on code replacement libraries available in your development environment, you can configure the code generator to replace instances of the following mutex and semaphore functions with application-specific implementations.

Function	Key
Mutex Destroy	`RTW_MUTEX_DESTROY`
Mutex Init	`RTW_MUTEX_INIT`
Mutex Lock	`RTW_MUTEX_LOCK`
Mutex Unlock	`RTW_MUTEX_UNLOCK`
Semaphore Destroy	`RTW_SEM_DESTROY`
Semaphore Init	`RTW_SEM_INIT`
Semaphore Post	`RTW_SEM_POST`
Semaphore Wait	`RTW_SEM_WAIT`

Operators

When generating C/C++ code from a Simulink model, depending on code replacement libraries available in your development environment, you can configure the code generator to replace instances of the following operators with application-specific implementations.

Mixed data type support indicates that you can specify different data types for different inputs.

Operator	Key	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
Addition (+)[1]	`RTW_OP_ADD`	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Subtraction (-)[1]	`RTW_OP_MINUS`	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Multiplication (*)[2]	`RTW_OP_MUL`	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Division (/)	`RTW_OP_DIV`	Integer Floating point Fixed-point Mixed	Scalar	Real Complex
Data type conversion (cast)	`RTW_OP_CAST`	Integer Floating point[3] Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Shift left (<<)	`RTW_OP_SL`	Integer Fixed-point Mixed	Scalar Vector Matrix[4]	Real

Operator	Key	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
Shift right arithmetic (>>)[5]	RTW_OP_SRA	Integer Fixed-point Mixed	Scalar Vector Matrix[4]	Real
Shift right logical (>>)	RTW_OP_SRL	Integer Fixed-point Mixed	Scalar Vector Matrix[4]	Real
Element-wise matrix multiplication (.*)[6]	RTW_OP_ELEM_MUL	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Matrix right division (/)	RTW_OP_RDIV	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Matrix left division (\)	RTW_OP_LDIV	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Matrix inversion (inv)	RTW_OP_INV	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Complex conjugation	RTW_OP_CONJUGATE	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Transposition (.')	RTW_OP_TRANS	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex

Operator	Key	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
Hermitian (complex conjugate) transposition (')	RTW_OP_HERMITIAN	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Multiplication with transposition[2]	RTW_OP_TRMUL	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Multiplication with Hermitian transposition[2]	RTW_OP_HMMUL	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Multiplication followed by shift right arithmetic (u1*u2>>u3)[7]	RTW_OP_MUL_SRA	Integer Fixed-point	Scalar	Real
Multiplication followed by division (u1*u2/u3)[8]	RTW_OP_MULDIV	Integer Fixed-point	Scalar	Real
Greater than (>)	RTW_OP_GREATER_THAN	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Greater than or equal (>=)	RTW_OP_GREATER_THAN_OR_EQUAL	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Less than (<)	RTW_OP_LESS_THAN	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex

Operator	Key	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
Less than or equal (<=)	RTW_OP_LESS_THAN_OR_EQUAL	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Equal (==)	RTW_OP_EQUAL	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Not equal (!=)	RTW_OP_NOT_EQUAL	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex

Operator	Key	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support

[1] See "Addition and Subtraction Operator Code Replacement" on page 51-170 for details to consider when defining mappings for addition and subtraction code replacements.

[2] Can map to Basic Linear Algebra Subroutine (BLAS) multiplication functions.

[3] Scaled floating point is not supported.

[4] Shift operator replacement with matrix data is supported for shift values that you specify with an input port. Replacement is not supported for shift values that you specify in a block parameter dialog.

[5] The code generator converts some arithmetic shift rights to logical shift rights. To avoid unexpected results, when creating a code replacement library that includes a table entry for an arithmetic shift right implementation, also include an entry for a logical shift right implementation.

[6] Use the multiplication (`*`) operator (`RTW_OP_MUL`) for scalar multiplication.

[7] Requires scalar, real, or fixed-point data types with zero bias; output type of the multiplication operation to accommodate all possible output values; shift operand is an unsigned integer; and net slope is equal to 1 (`U1_slope * U2_slope == Mul_output_slope` and `Mul_output_slope == output_slope_of_shift_operation`).

[8] Requires scalar, real, or fixed-point data types with zero bias; output type of the multiplication operation to accommodate all possible output values; and net slope is equal to 1 (`U1_slope * U2_slope == Mul_output_slope == U3_slope * Div_output_slope`).

More About

- "Lookup Table Function Code Replacement" on page 51-112
- "Develop a Code Replacement Library" on page 51-27
- "Quick Start Library Development" on page 51-28

- “What Is Code Replacement?” on page 37-2

Develop a Code Replacement Library

Iterate through the following steps, as necessary, to develop a code replacement library:

1. "Identify Code Replacement Requirements" on page 51-38
2. "Prepare for Code Replacement Library Development" on page 51-41
3. "Define Code Replacement Mappings" on page 51-42
4. "Specify Build Information for Replacement Code" on page 51-59
5. "Register Code Replacement Mappings" on page 51-68
6. "Verify Code Replacements" on page 51-76
7. "Deploy Code Replacement Library" on page 51-93

To get started, see "Identify Code Replacement Requirements" on page 51-38.

To experiment with the process and tools, see "Quick Start Library Development" on page 51-28.

More About

- "Code You Can Replace From Simulink Models" on page 51-7
- "Identify Code Replacement Requirements" on page 51-38
- "Quick Start Library Development" on page 51-28
- "What Is Code Replacement Customization?" on page 51-3

Quick Start Library Development

This example shows how to develop a code replacement library that includes an entry for generating replacement code for the math function `sin`. You use the Code Replacement Tool.

Prerequisites

To complete this example, install the following software:

- MATLAB
- Simulink
- Simulink Coder
- Embedded Coder

For instructions on installing MathWorks products, see the "Installation and Activation" (Installation, Licensing, and Activation). If you have installed MATLAB and want to see what other MathWorks products are installed, in the Command Window, enter `ver`.

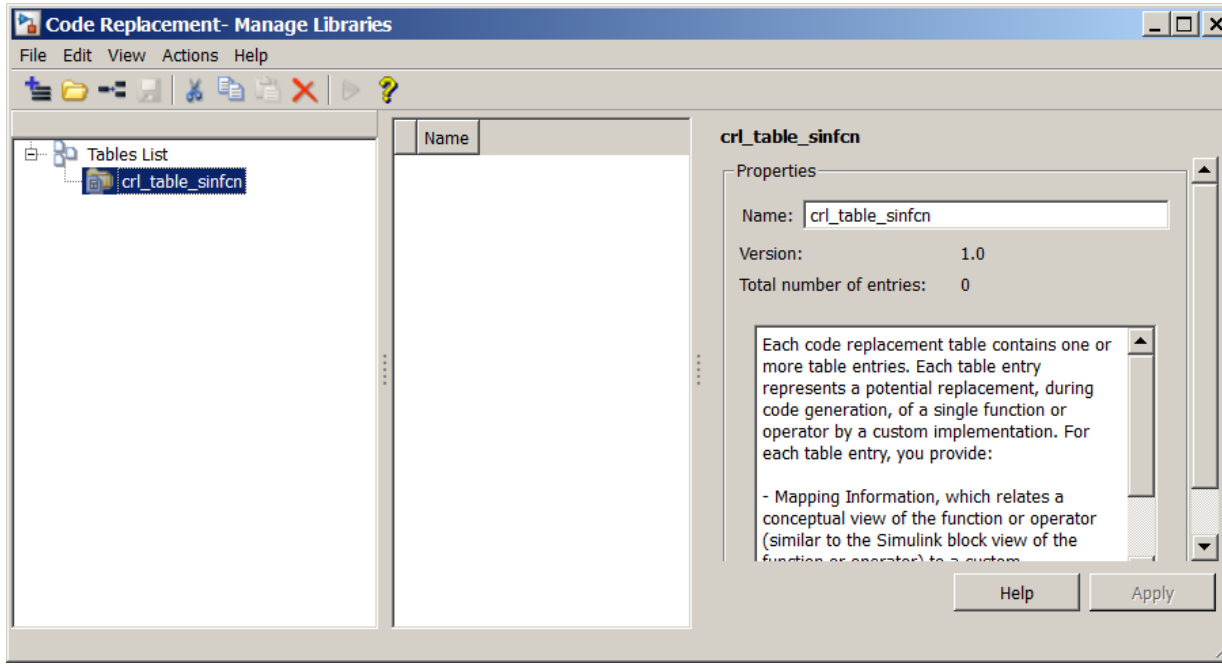
For a list of supported compilers, see http://www.mathworks.com/support/compilers/current_release/.

Open the Code Replacement Tool

1. Start a new MATLAB session.
2. Create or navigate (`cd`) to an empty folder.
3. At the command prompt, enter the `crtool` command. The Code Replacement Tool window opens.

Create Code Replacement Table

1. In the Code Replacement Tool window, select **File > New table**.
2. In the right pane, name the table `crl_table_sinfcn` and click **Apply**. Later, when you save the table, the tool saves it with the file name `crl_table_sinfcn.m`.



Create Table Entry

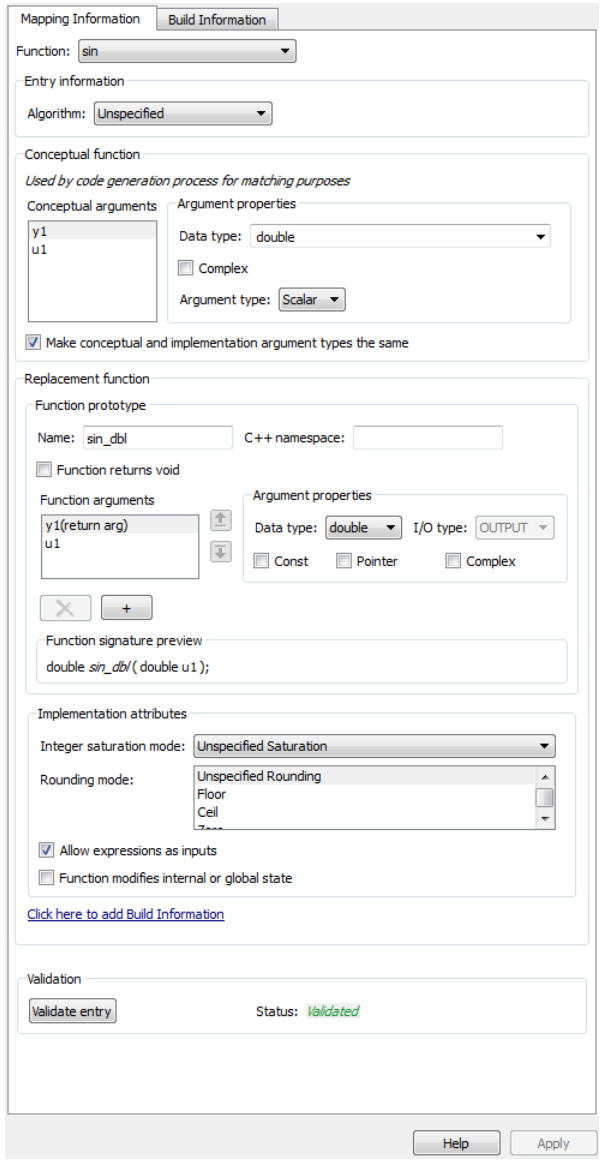
Create a table entry that maps a `sin` function with `double` input and `double` output to a custom implementation function.

1. In the left pane, select table `crl_table_sinfcn`. Then, select **File** > **New entry** > **Function**. The new entry appears in the middle pane, initially without a name.
2. In the middle pane, select the new entry.
3. In the right pane, on the **Mapping Information** tab, from the **Function** menu, select `sin`.
4. Leave **Algorithm** set to `Unspecified`, and leave parameters in the **Conceptual function** group set to default values.
5. In the **Replacement function** group, name the replacement function `sin_dbl`.
6. Leave the remaining parameters in the **Replacement function** group set to default values.

7 Click **Apply**. The tool updates the **Function signature preview** to reflect the specified replacement function name.

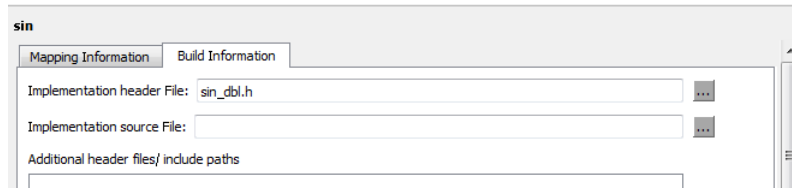
8 Scroll to the bottom of the **Mapping Information** tab and click **Validate entry**. The tool validates your entry.

The following figure shows the completed mapping information.


Mapping Information
Build Information
Function: sin
Entry information
Algorithm: Unspecified
Conceptual function
Used by code generation process for matching purposes
Conceptual arguments
y1
u1
Argument properties
Data type: double
Complex
Argument type: Scalar
Make conceptual and implementation argument types the same
Replacement function
Function prototype
Name: sin_dbl
C++ namespace:
Function returns void
Function arguments
y1(return arg)
u1
Argument properties
Data type: double
I/O type: OUTPUT
Const
Pointer
Complex
+
Function signature preview
double sin_dbl(double u1);
Implementation attributes
Integer saturation mode: Unspecified Saturation
Rounding mode:
Unspecified Rounding
Floor
Ceil
Allow expressions as inputs
Function modifies internal or global state
Click here to add Build Information
Validation
Validate entry
Status: Validated
Help
Apply

Specify Build Information for Replacement Code

1. On the **Build Information** tab, for the **Implementation header file** parameter, enter `sin_dbl.h`.
2. Leave the remaining parameters set to default values.
3. Click **Apply**.



4. Optionally, you can revalidate the entry. Return to the **Mapping Information** tab and click **Validate entry**.

Create Another Table Entry

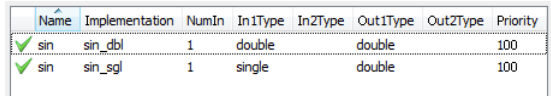
Create an entry that maps a `sin` function with `single` input and `double` output to a custom implementation function named `sin_sgl`. Create the entry by copying and pasting the `sin_dbl` entry.

1. In the middle pane, select the `sin_dbl` entry.
2. Select **Edit** > **Copy**
3. Select **Edit** > **Paste**
4. On the **Mapping Information** tab, in the **Conceptual function** section, set the data type of input argument `u1` to `single`.
5. In the **Replacement function** section, name the function `sin_sgl`. Set the data type of input argument `u1` to `single`.
6. Click **Apply**. Note the changes that appear for the **Function signature preview**.
7. On the **Build Information** tab, for the **Implementation header file** parameter, enter `sin_sgl.h`. Leave the remaining parameters set to default values and click **Apply**.

Validate the Code Replacement Table

1. Select **Actions** > **Validate table**.

2 If the tool reports errors, fix them, and rerun the validation. Repeat fixing and validating errors until the tool does not report errors. The following figure shows a validation report.



	Name	Implementation	NumIn	In1Type	In2Type	Out1Type	Out2Type	Priority
✓	sin	sin_dbl	1	double		double		100
✓	sin	sin_sgl	1	single		double		100

Save the Code Replacement Table

Save the code replacement table to a MATLAB file in your working folder. Select **File** > **Save table**. By default, the tool uses the table name to name the file. For this example, the tool saves the table in the file `crl_table_sinfcn.m`.

Review the Code Replacement Table Definition

Consider reviewing the MATLAB code for your code replacement table definition. After using the tool to create an initial version of a table definition file, you can update, enhance, or copy the file in a text editor.

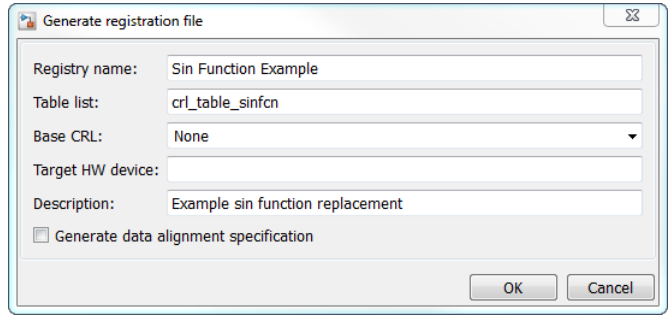
To review it, in MATLAB or another text editor, open the file `crl_table_sinfcn.m`.

Generate a Registration File

Before you can use your code replacement table, you must register it as part of a code replacement library. Use the Code Replacement Tool to generate a registration file.

1 In the Code Replacement Tool, select **File** > **Generate registration file**.

2 In the **Generate registration file** dialog box, edit the dialog box fields to match the following figure, and then click **OK**.



3 In the **Select location** dialog box, specify a location for the registration file. The location must be on the MATLAB path or in the current working folder. Save the file. The tool saves the file as `rtwTargetInfo.m`.

Register the Code Replacement Table

At the command prompt, enter:

```
sl_refresh_customizations
```

Review and Test Code Replacements

Apply your code replacement library. Verify that the code generator makes code replacements that you expect.

1 Check for errors. At the command line, invoke the table definition file. For example:

```
tbl = crl_table_sinfcn

tbl =

  TflTable with properties:


               Version: '1.0'
       ReservedSymbols: []
    StringResolutionMap: []
             AllEntries: [2x1 RTW.TflCFunctionEntry]
           EnableTrace: 1
```

If an error exists in the definition file, the invocation triggers a message. Fix the error and try again.

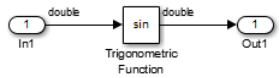
2 Use the Code Replacement Viewer to check your code replacement entries. For example:

```
crviewer('Sin Function Example')
```

In the viewer, select entries in your table and verify that the content is what you expect. The viewer can help you detect issues such as:

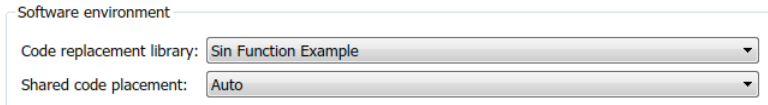
- Incorrect argument order.
- Conceptual argument names that do not match what the code generator expects.
- Incorrect priority settings.

3 Identify an existing model or create a new model that includes a Trigonometric Function block that is set to the `sin` function. For example:



4 Open the model and configure it for code generation with an Embedded Coder (ERT-based) target.

5 See whether your library is listed as an available option for the **Code Generation > Interface > Code replacement library** model configuration parameter. If it is, select it.



If it is not listed, open the registration file, `rtwTargetInfo.m`. See whether you entered the correct code replacement table name when you created the file. If you hover the cursor over the selected library, a tool tip appears. This tip contains information derived from your code replacement library registration file, such as the library description and the list of tables it contains.

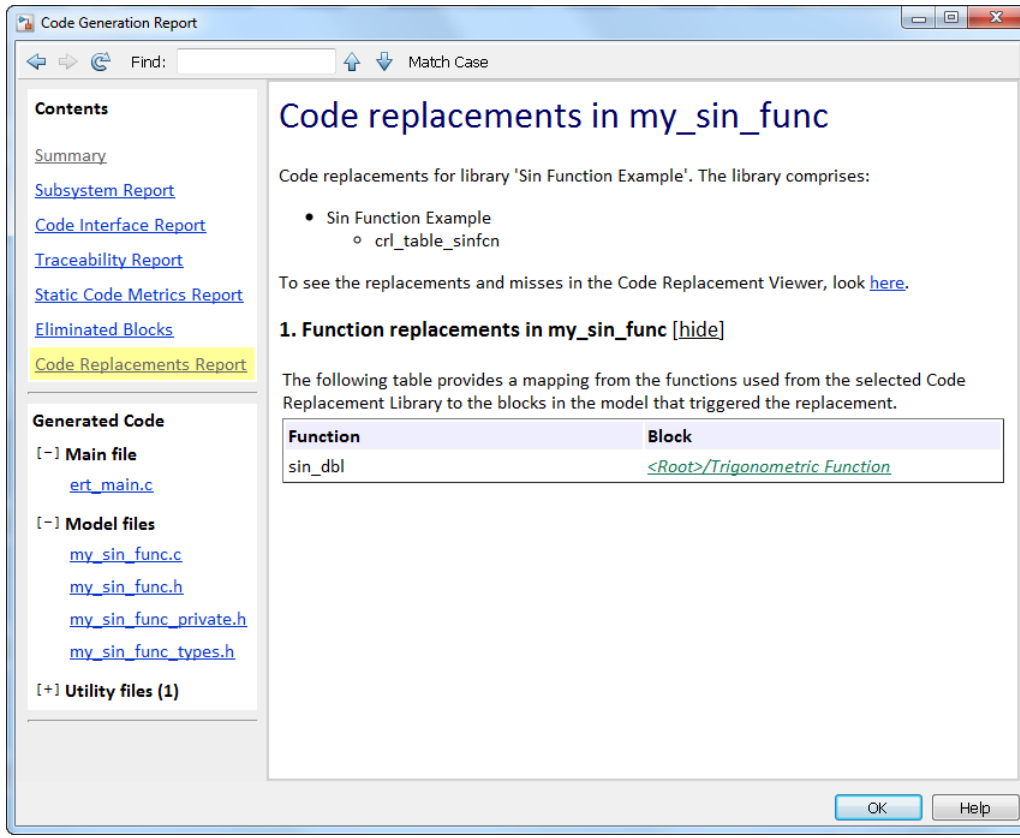
6 Configure the code generation report for code replacement analysis by setting the following parameters:

- On the **Code Generation > Report** pane, select **Create code generation report** and **Open report automatically**.
- On the **Code Generation > Comments** pane, select **Include comments**, **Simulink block / Stateflow object comments**, and **Simulink block descriptions**.
- On the **All Parameters** tab, select **Code-to-model**, **Model-to-code**, and **Summarize which blocks triggered code replacements**.

7 Configure the model to generate code only. Before you build an executable, confirm that the code generator is replacing code as expected.

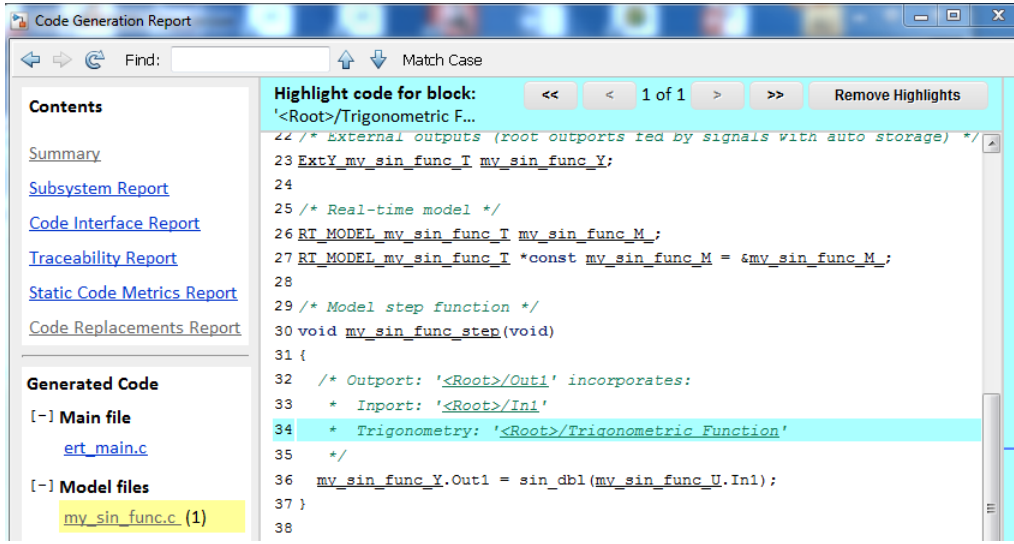
8 Generate code for the model.

9 Review code replacement results in the Code Replacement Report section of the code generation report.



The report indicates that the code generator found a match and applied the replacement code for the function `sin_dbl`.

10 Review the code replacements. In the model window, right-click the Trigonometric Function block. Select **C/C++ Code > Navigate to C/C++ Code**. The code generation report opens and highlights the code replacement in `my_sin_func.c`. In this case, the code generator replaced `sin` with `sin_dbl`.



More About

- "Code You Can Replace From Simulink Models" on page 51-7
- "Develop a Code Replacement Library" on page 51-27
- "What Is Code Replacement Customization?" on page 51-3

Identify Code Replacement Requirements

The first step to developing a code replacement library is to consider the following types of requirements for the library.

Mapping Information Requirements

- Are you defining a code replacement mapping for the first time?
- Are you updating code replacement entries in an existing library? Or, are you creating a new library?
- Are you rapid prototyping code replacements?
- Can you base your mappings on existing mappings?
- What type of code do you want to replace? Options include:
  - Math operation
  - Function
  - BLAS operation
  - CBLAS operation
  - Net slope fixed-point operation
  - Semaphore or mutex functions
- Do you want to change the inline or nonfinite behavior for functions?
- What specific functions and operations do you want to replace?
- What input and output arguments does the function or operator that you are replacing take? For each argument, what is the data type, complexity, and dimensionality?
- What does the prototype for your replacement code look like?
  - What is the replacement function name?
  - What are the input and output arguments?
  - Are there return values?
  - What is the data type, complexity, and dimensionality of each argument and return value?

Build Information Requirements

- Does your replacement function implementation require a header file? If yes, specify the header file.
- If the replacement function implementation requires a header file, what is the path for that file?
- Is the source file for the replacement function in your working folder? If not, you can explicitly specify the source file name and extension. For example, if the file is required in the generated makefile or specified in a build information object, specify the source file.
- Does the replacement function use additional include files? If yes, what are they and what are the paths for those files?
- Does the replacement function use additional source files? If yes, what are they and what are the paths for those files?
- What compiler flags are required for compiling code that includes the replacement code?
- What linker flags are required for building an executable that includes the replacement code?
- Are the required header, source, and object files for building an executable that includes your replacement code in the working folder for your project? If not, before starting the build process, do you want the code generator to copy required files to the build folder?

Registration Information Requirements

- What do you want to name your code replacement library?
- What code replacement tables do you want to include in the library? What are the file names and paths for the tables?
- What is the purpose of the library? You can document the purpose as the library description.
- Does the library apply to specific hardware devices? If yes, what devices?
- Are you developing a hierarchy of code replacement libraries? Is the library that you are developing based (dependent) on another library? For example, you can specify a general `TI device library` as the base library for a more specific `TI C28x` device library.

- Do you need to specify data alignment for the library? What data alignments are required? For each specification, what type of alignment is required and for what programming language?

Next, prepare for developing a library by reviewing a code replacement library development checklist.

More About

- "Code You Can Replace From Simulink Models" on page 51-7
- "Develop a Code Replacement Library" on page 51-27
- "Prepare for Code Replacement Library Development" on page 51-41
- "What Is Code Replacement Customization?" on page 51-3

Prepare for Code Replacement Library Development

After you identify your code replacement requirements, prepare for library development by reviewing this checklist:

- Get familiar with the library development process.
- Decide whether to define code replacement mappings and produce a registration file interactively with the Code Replacement Tool or programmatically.
- Identify or develop MATLAB code and Simulink models to test your code replacement library.
- Consider the hierarchy and organization of your library. A library can consist of multiple tables and each table can include multiple entries. How do you want to organize the library to optimize reuse of tables and entries? For example, a registration file can define code replacement tables organized in a hierarchy of code replacement libraries based on entries that increase in specificity:
  - Common entries
  - Entries for TI devices
  - Entries for TI C6xx devices
  - Entries specific to the TI C67x device
- If support files, such as header files, additional source files, and dynamically linked libraries are not in your current working folder, note their location. You need to specify the paths for such files.

Next, based on your requirements and preparation, define code replacement mappings.

More About

- “Code You Can Replace From Simulink Models” on page 51-7
- “Identify Code Replacement Requirements” on page 51-38
- “Define Code Replacement Mappings” on page 51-42
- “Develop a Code Replacement Library” on page 51-27
- “What Is Code Replacement Customization?” on page 51-3

Define Code Replacement Mappings

After you prepare for library development, use your requirements to define code replacement mappings. A code replacement mapping associates a conceptual representation of a function or operator that is familiar to the code generator with a custom implementation representation that specifies a C or C++ replacement function prototype. You capture a mapping as an entry in a code replacement table:

- Interactively, by using the Code Replacement Tool.
- Programmatically, by using a MATLAB programming interface.

Choose an Approach for Defining Code Replacement Mappings

The following table lists situations to help you decide when to use the interactive or programmatic approach.

Situation	Approach
Defining mappings for the first time.	Code Replacement Tool.
Rapid prototyping mappings.	Code Replacement Tool to quickly generate, register, and test mappings.
Developing a mapping as a template or starting point for defining similar mappings.	Code Replacement Tool to generate definition code that you can copy and modify.
Modifying a registration file, including copying and pasting content.	MATLAB Editor to update the programming interface directly.
Defining mappings that specify attributes not available from the Code Replacement Tool (for example, sets of algorithm parameters).	Programming interface.
Reusing existing code for new mappings by copying, pasting, and editing existing mappings.	Programming interface.

Define Mappings Interactively with the Code Replacement Tool

This example shows how to use the Code Replacement Tool to develop code replacement mappings. The tool is ideal for getting started with developing mappings, rapid prototyping, and developing a mapping to use as a starting point for defining similar mappings.

Open the Code Replacement Tool

Do one of the following:

- In the Command Window, enter the command `crtool`.
- In the Configuration Parameters dialog box, navigate to **All Parameters** > **Code Generation** > **Code replacement library** and click **Custom**.

An Embedded Coder license is not required to create a custom code replacement library. However, you must have an Embedded Coder license to use a such a library.

By default, the tool displays, left to right, a root pane, a list pane, and a dialog pane. You can manipulate the display:

- Drag boundaries to widen, narrow, shorten, or lengthen panes, and to resize table columns.
- Select **View** > **Show dialog pane** to hide or display the right-most pane.
- Click a table column heading to sort the table based on contents of the selected column.
- Right-click a table column heading and select **Hide** to remove the column from the display. (You cannot hide the **Name** column.)

Create a Code Replacement Table

1. In the Code Replacement Tool window, select **File** > **New table**.
2. In the right pane, name the table and click **Apply**. Later, when you save the table, the tool uses the table name that you specify to name the file. For example, if you enter the name `my_sinfcn`, the tool names the file `my_sinfcn.m`.

Create Table Entries

Create one or more table entries. Each entry maps the conceptual representation of a function or operator to your implementation representation. The information that you enter depends on the type of entry you create. Enter the following information:

1. In the left pane, select the table to which you want to add the entry.
2. Select **File** > **New entry** > **entry-type**, where **entry-type** is one of:

   - Math Operation
   - Function
   - BLAS Operation
   - CBLAS Operation
   - Net Slope Fixed-Point Operation
   - Semaphore entry
   - Customization entry

   The new entry appears in the middle pane, initially without a name.
3. In the middle pane, select the new entry.
4. In the right pane, on the **Mapping Information** tab, from the **Function** or **Operation** menu, select the function or operation that you want the code generator to replace. Regardless of the entry type, make a selection from this menu. Your selection determines what other information you specify.

   Except for customization entries, you also specify information for your replacement function prototype. You can also specify implementation attributes, such as the rounding modes to apply.
5. If prompted, specify additional entry information that you want the code generator to use when searching for a match. For example, when you select an addition or subtraction operation, the tool prompts you to specify an algorithm (`Cast before operation` or `Cast after operation`).
6. Review the conceptual argument information that the tool populates for the function or operation. Conceptual input and output arguments represent arguments for the function or operator being replaced. Conceptual arguments observe naming conventions (`'y1'`, `'u1'`, `'u2'`, ...) and data types familiar to the code generator.

   If you do not want the data types for your implementation to be the same as the conceptual argument types, clear the **Make the conceptual and implementation argument types the same** check box. For example, most ANSI-C functions operate on and return `double` data. Clear the check box if want to map a conceptual representation of the function to an implementation representation that specifies an argument and return value. For example, clear the check box to map the conceptual representation of the function `sin` to an implementation representation that

specifies an argument and return value of type `single` (`single sin(single)`), of type `double` (`double sin(double)`. In this case, the code generator produces the following code:

```
y = (single) sin(u1);
```

If you select `Custom` for a function entry, specify only conceptual argument information.

7 Specify the name and argument information for your replacement function. As you enter the information and click **Apply**, the tool updates the **Function signature preview**.

8 Specify additional implementation attributes that apply. For example, depending on the type and name of the entry that you specify, the tool prompts you to specify:

- Integer saturation mode
- Rounding modes
- Whether to allow inputs that include expressions
- Whether a function modifies internal or global state

9 Click **Apply**.

Validate Tables and Entries

The Code Replacement Tool provides a way to validate the syntax of code replacement tables and table entries as you define them. If the tool finds validation errors, you can address them and retry the validation. Repeat the process until the tool does not report errors.

To	Do
Validate table entries	Select an entry, scroll to the bottom of the **Mapping Information** tab, and click **Validate entry**. Alternatively, select one or more entries, right-click, and select **Validate entries**.
Validate a table	Select the table. Then, select **Actions** > **Validate table**.

Save a Table

When you save a table, the tool validates unvalidated content.

1. Select **File** > **Save table**.
2. In the Browse For Folder dialog box, specify a location and name for the file. Typically, you select a location on the MATLAB path. By default, the tool names the file using the name that you specify for the table with the extension `.m`.
3. Click **Save**.

Open and Modify Tables

After saving a code replacement table, to make changes in the table:

1. Select **File** > **Open table**.
2. In the Import file dialog box, browse to the MATLAB file that contains the table.

Repeat the sequence to open and work on multiple tables.

If you open multiple tables, you can manage the tables together. For example, use the tool to:

- Create new table entries.
- Delete entries.
- Copy and paste or cut and paste information between tables.

Define Mappings Programmatically

This example shows how to define a code replacement mapping programmatically. The programming interface for defining code replacement table mappings is ideal for

- Modifying tables that you create with the Code Replacement Tool.
- Defining mappings for specialized entries that you cannot create with the Code Replacement Tool.
- Replicating and modifying similar entries and tables.

Steps for defining a mapping programmatically are:

Create Code Replacement Table

1. Create a table definition file that contains a function definition. For example:

   ```
   function hTable = crl_table_sinfcn()
   ```
2. Within the function body, create the table by calling the function `RTW.TflTable`.

```
hTable = RTW.TflTable;
```

Create Table Entry

For each function or operator that you want the code generator to replace, map a conceptual representation of the function or operator to an implementation representation as a table entry.

1 Within the body of a table definition file, create a code replacement entry object. Call one of the following functions.

Entry Type	Function
Math operation	`RTW.TflCOperationEntry`
Function	`RTW.TflCFunctionEntry`
BLAS operation	`RTW.TflBlasEntryGenerator`
CBLAS operation	`RTW.TflCBlasEntryGenerator`
Fixed-point addition and subtraction operations (support for `SlopesMustBeTheSame` and `ZeroNetBias` parameters)	`RTW.TflCOperationEntryGenerator`
Net slope fixed-point operation	`RTW.TflCOperationEntryGenerator_NetSlope`
Semaphore or mutex entry	`RTW.TflCSemaphoreEntry`
Custom function entry	*`MyCustomFunctionEntry`* (where *`MyCustomFunctionEntry`* is a class derived from `RTW.TflCFunctionEntryML`)
Custom operation entry	*`MyCustomOperationEntry`* (where *`MyCustomOperationEntry`* is a class derived from `RTW.TflCOperationEntryML`)

For example:

```
hEnt = RTW.TflCFunctionEntry;
```

You can combine steps of creating the entry, setting entry parameters, creating conceptual and implementation arguments, and adding the entry to a table with a

single function call to `registerCFunctionEntry`, `registerCPPFunctionEntry`, or `registerCPromotableMacroEntry` if you are creating an entry for a function and the function implementation meets the following criteria:

- Implementation argument names and order match the names and order of corresponding conceptual arguments.
- Input arguments are of the same type.
- The return and input argument names follow the code generator's default naming conventions:
  - Return argument is `y1`.
  - Input arguments are `u1`, `u2`, ..., `un`.

For example:

```
registerCFunctionEntry(hTable, 100, 1, 'sin', 'double', ...
   'sin_dbl', 'double', 'sin_dbl.h','','');
```

As another alternative, you can significantly reduce the amount of code that you write by combining the steps of creating the entry and conceptual and implementation arguments with a call to the `createCRLEntry` function. In this case, specify the conceptual and implementation information as character vector specifications.

For example:

```
hEnt = createCRLEntry(hTable, ...
    'double y1 = sin(double u1)', ...
    'mySin');
```

This approach does not support:

- C++ implementations
- Data alignment
- Operator replacement with net slope arguments
- Entry parameter specifications (for example, priority, algorithm, building information)
- Semaphore and mutex function replacements

Set Entry Parameters

Set entry parameters, such as the priority, algorithm information, and implementation (replacement) function name. Call the function listed in the following table for the entry type that you created.

Entry Type	Function
Math operation	`setTflCOperationEntryParameters`
Function	`setTflCFunctionEntryParameters`
BLAS operation	`setTflCOperationEntryParameters`
CBLAS operation	`setTflCOperationEntryParameters`
Fixed-point addition and subtraction operations where there is a many-to-one mapping, such as a mapping for a range of fixed-point types to the same replacement function (support for `SlopesMustBeTheSame` and `ZeroNetBias` parameters)	`setTflCOperationEntryParameters`
Net slope fixed-point operation	`setTflCOperationEntryParameters`
Semaphore or mutex entry	`setTflCSemaphoreEntryParameters`
Custom function entry	`setTflCFunctionEntryParameters`
Custom operation entry	`setTflCOperationEntryParameters`

To see a list of the parameters that you can set, at the command line, create a new entry and omit the semicolon at the end of the command. For example:

```
hEnt = RTW.TflCFunctionEntry
```

```
hEnt =

  TflCFunctionEntry with properties:

             Implementation: [1x1 RTW.CImplementation]
        SlopesMustBeTheSame: 0
          BiasMustBeTheSame: 0
            AlgorithmParams: []
                   ImplType: 'FCN_IMPL_FUNCT'
      AdditionalHeaderFiles: {0x1 cell}
      AdditionalSourceFiles: {0x1 cell}
```

```
      AdditionalIncludePaths: {0x1 cell}
       AdditionalSourcePaths: {0x1 cell}
          AdditionalLinkObjs: {0x1 cell}
      AdditionalLinkObjsPaths: {0x1 cell}
          AdditionalLinkFlags: {0x1 cell}
       AdditionalCompileFlags: {0x1 cell}
                  SearchPaths: {0x1 cell}
                          Key: ''
                     Priority: 100
               ConceptualArgs: [0x1 handle]
                    EntryInfo: []
                   GenCallback: ''
                   GenFileName: ''
               SaturationMode: 'RTW_SATURATE_UNSPECIFIED'
                RoundingModes: {'RTW_ROUND_UNSPECIFIED'}
           TypeConversionMode: 'RTW_EXPLICIT_CONVERSION'
              AcceptExprInput: 1
                  SideEffects: 0
                   UsageCount: 0
           RecordedUsageCount: 0
                  Description: ''
    StoreFcnReturnInLocalVar: 0
                TraceManager: [1x1 RTW.TflTraceManager]
```

To see the implementation parameters, enter:

```
hEnt.Implemenation
```

```
ans =

  CImplementation with properties:

              HeaderFile: ''
              SourceFile: ''
              HeaderPath: ''
              SourcePath: ''
                  Return: []
          StructFieldMap: []
                    Name: ''
               Arguments: [0x1 handle]
    ArgumentDescriptor: []
```

For example, to set entry parameters for the `sin` function and name your replacement function `sin_dbl`, use the following function call:

```
setTflCFunctionEntryParameters(hEnt, ...
          'Key', 'sin', ...
          'ImplementationName', 'sin_dbl');
```

Create Conceptual Arguments

Create conceptual arguments and add them to the entry's array of conceptual arguments.

- Specify output arguments before input arguments.
- Specify argument names that comply with code generator argument naming conventions:
  - `y1` for a return argument
  - `u1`, `u2`, ..., `un` for input arguments
- Specify data types that are familiar to the code generator.
- The function signature, including argument naming, order, and attributes, must fulfill the signature match sought by function or operator callers.
- The code generator determines the size of the value for an argument with an unsized type, such as integer, based on hardware implementation configuration settings.

For each argument:

1. Identify whether the argument is for input or output, the name, and data type. If you do not know what arguments to specify for a supported function or operation, use the Code Replacement Tool to find them. For example, to find the conceptual arguments for the `sin` function, open the tool, create a table, create a function entry, and in the **Function** menu select `sin`.
2. Create and add the conceptual argument to an entry. You can choose a method from the methods listed in this table.

If	Then

You want simpler code or want to explicitly specify whether the argument is scalar or nonscalar (vector or matrix).	Call the function `createAndAddConceptualArg`. For example:
	`createAndAddConceptualArg(hEnt, ...`
	`  'RTW.TflArgNumeric', ...`
	`  'Name', 'y1',...`
	`  'IOType', 'RTW_IO_OUTPUT',...`
	`  'DataTypeMode', 'double');`

If	Then
	The second argument specifies whether the argument is scalar (`RTW.TflArgNumeric` or`RTW.TflArgMatrix`) .

You want to create an argument based on a built-in argument definition (for example, scalar or nonscalar).	Call `getTflArgFromString` to create the argument. Then, call `addConceptualArg` to add the argument to the entry.
	`arg = getTflArgFromString(hEnt, 'y1','double');`
	`arg.IOType = 'RTW_IO_OUTPUT';`
	`addConceptualArg(hEnt, arg);`

You need to define several similar mappings, you want to minimize the code to write, and the entries do not require data alignment, use net slope arguments, or involve semaphore or mutex replacements.	Call `createCRLEntry` to create the entry and specify conceptual and implementation arguments in a single function call.
	`hEnt = createCRLEntry(hTable, ...`
	`'double y1 = sin(double u1)', ...`
	`'mySin');`

The following code shows the second approach listed in the table for specifying the conceptual output and input argument definitions for the `sin` function.

```
% Conceptual Args

arg = getTflArgFromString(hEnt, 'y1','double');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEnt, arg);

arg = getTflArgFromString(hEnt, 'u1','double');
addConceptualArg(hEnt, arg);
```

Create Implementation Arguments

Create implementation arguments for the C or C++ replacement function and add them to the entry.

- When replacing code, the code generator uses the argument names to determine how it passes data to the implementation function.

- For function replacements, the order of implementation argument names must match the order of the conceptual argument names.
- For operator replacements, the order of implementation argument names do not have to match the order of the conceptual argument names. For example, for an operator replacement for addition, `y1=u1+u2`, the conceptual arguments are `y1`, `u1`, and `u2`, in that order. If the signature of your implementation function is `t myAdd(t u2, t u1)`, where `t` is a valid C type, based on the argument name matches, the code generator passes the value of the first conceptual argument, `u1`, to the second implementation argument of `myAdd`. The code generator passes the value of the second conceptual argument, `u2`, to the first implementation argument of `myAdd`.
- For operator replacements, you can remap operator output arguments to implementation function input arguments.

For each argument:

1. Identify whether the argument is for input or output, the name, and the data type.
2. Create and add the implementation argument to an entry. You can choose a method from the methods listed in this table.

If	Then
You want to populate implementation arguments as copies of previously created matching conceptual arguments	Call the function `copyConceptualArgsToImplementation`. For example: `copyConceptualArgsToImplementation(hEnt);`

You want to create and add implementation arguments individually, or vary argument attributes, while maintaining conceptual argument order	Call functions `createAndSetCImplementationReturn` and`createAndAddImplementationArg` . For example:
	`createAndSetCImplementationReturn(hEnt,`
	`'RTW.TflArgNumeric', ...`
	`'Name', 'y1', ...`
	`'IOType', 'RTW_IO_OUTPUT', ...`
	`'IsSigned', true, ...`
	`'WordLength', 32, ...`
	`'FractionLength', 0);`
	
	`createAndAddImplementationArg(op_entry,`
	`'RTW.TflArgNumeric',...`
	`'Name', 'u1', ...`

If	Then

	`'IOType', 'RTW_IO_INPUT',...`
	`'IsSigned', true,...`
	`'WordLength', 32, ...`
	`'FractionLength', 0 );`

If	Then

You want to minimize the amount of code, or specify constant arguments to pass to the implementation function	Create the argument with a call to the function `getTflArgFromString`. Then, use the convenience method `setReturn` or `addArgument` to specify whether an argument is a return value or argument and to add the argument to the entry's array of implementation arguments. For example:
	
	`arg = getTflArgFromString(hEnt, 'y1','double');`
	`arg.IOType = 'RTW_IO_OUTPUT';`
	`hEnt.Implementation.setReturn(arg);`
	
	`arg = getTflArgFromString(hEnt, 'u1','double');`
	`hEnt.Implementation.addArgument(arg);`
	
	The following call to `getTflArgFromString` passes the constant `0` to argument `u2`:
	
	`arg = getTflArgFromString(hEnt, 'u2', 'int16', 0)`
	`hEnt.Implementation.addArgument(arg);`
	
	For semaphore and mutex entries, use the functions `getTflDWorkFromString` and `addDWorkArg` to create and add a DWork argument to the entry. Then create implementation arguments as shown above with `getTflArgFromString` and the convenience methods `setReturn` and `addArgument`. For example:
	
	`arg = getTflDWorkFromString('d1', 'void*')`
	`hEnt.addDWorkArg(arg);`
	
	`arg = hEnt.getTflArgFromString('y1', 'void');`
	`arg.IOType = 'RTW_IO_OUTPUT';`
	`hEnt.Implementation.setRetrurn(arg);`
	
	`arg = hEnt.getTflArgFromString('u1', 'integer');`
	`hEnt.Implementation.addArgument(arg);`
	
	`arg = hEnt.getTflArgFromString('d1', 'void**');`
	`hEnt.Implementation.addArgument(arg);`

If	Then

You need to define several similar mappings, you want to minimize the code to write, and the entries do not require data alignment, use net slope arguments, or involve semaphore or mutex replacements.	Call `createCRLEntry` to create the entry and specify conceptual and implementation arguments in a single function call.
	`hEnt = createCRLEntry(hTable, ...`
	`'double y1 = sin(double u1)', ...`
	`'mySin');`

The following code shows the third approach listed in the table for specifying the implementation output and input argument definitions for the `sin` function:

```
% Implementation Args

arg = hEnt.getTflArgFromString('y1','double');
arg.IOType = 'RTW_IO_OUTPUT';
hEnt.Implementation.setReturn(arg);

arg = hEnt.getTflArgFromString('u1','double');
hEnt.Implementation.addArgument(arg);
```

Add Entry to Table

Add an entry to a code replacement table by calling the function `addEntry`.

```
addEntry(hTable, hEnt);
```

Validate Entry

After you create or modify a code replacement table entry, validate it by invoking it at the MATLAB command line. For example:

```
hTbl = crl_table_sinfcn

hTbl =

RTW.TflTable
    Version: '1.0'
    AllEntries: [2x1 RTW.TflCFunctionEntry]
    ReservedSymbols: []
```

```
    StringResolutionMap: []
```

If the table includes errors, MATLAB reports them. The following examples shows how MATLAB reports a typo in a data type name:

```
hTbl = crl_table_sinfcn

??? RTW_CORE:tfl:TflTable: Unsupported data type, 'dooble'.

Error in ==> crl_table_sinfcn at 7
hTable.registerCFunctionEntry(1OO, 1, 'sin', 'dooble', 'sin_dbl', ...
```

Save Table

Save the table definition file. Use the name of the table definition function to name the file, for example, `crl_table_sinfcn.m`.

Next, from your requirements, determine whether you need to specify build information for your replacement code.

More About

- "Code You Can Replace From Simulink Models" on page 51-7
- "Math Function Code Replacement" on page 51-94
- "Memory Function Code Replacement" on page 51-96
- "Nonfinite Function Code Replacement" on page 51-99
- "Semaphore and Mutex Function Replacement" on page 51-102
- "Algorithm-Based Code Replacement" on page 51-109
- "Lookup Table Function Code Replacement" on page 51-112
- "Data Alignment for Code Replacement" on page 51-133
- "Replace MATLAB Functions with Custom Code Using coder.replace" on page 51-142
- "Replace MATLAB Functions Specified in MATLAB Function Blocks" on page 51-148
- "Customize Match and Replacement Process" on page 51-153
- "Scalar Operator Code Replacement" on page 51-168
- "Addition and Subtraction Operator Code Replacement" on page 51-170
- "Small Matrix Operation to Processor Code Replacement" on page 51-174
- "Matrix Multiplication Operation to MathWorks BLAS Code Replacement" on page 51-178

- "Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement" on page 51-186
- "Remap Operator Output to Function Input" on page 51-192
- "Customize Code Match and Replacement for Scalar Operations" on page 51-161
- "Fixed-Point Operator Code Replacement" on page 51-195
- "Binary-Point-Only Scaling Code Replacement" on page 51-203
- "Slope Bias Scaling Code Replacement" on page 51-207
- "Net Slope Scaling Code Replacement" on page 51-211
- "Equal Slope and Zero Net Bias Code Replacement" on page 51-218
- "Data Type Conversions (Casts) and Operator Code Replacement" on page 51-222
- "Shift Left Operations and Code Replacement" on page 51-226
- "Optimize Generated Code By Developing and Using Code Replacement Libraries - Simulink®"
- "Prepare for Code Replacement Library Development" on page 51-41
- "Specify Build Information for Replacement Code" on page 51-59
- "Develop a Code Replacement Library" on page 51-27
- "What Is Code Replacement Customization?" on page 51-3

Specify Build Information for Replacement Code

After you define code replacement mappings, determine whether you need to specify build information for your replacement code. A code replacement table entry can specify build information for the code generator to use when replacing code for a match. For example, specify files for implementation replacement code if you are using a generated makefile and the code generation software compiles the code.

Add build information to an entry:

- Interactively, by using the **Build Information** tab in the Code Replacement Tool.
- Programmatically, by using a MATLAB programming interface.

Build Information

The build information can include:

- Paths and file names for header files
- Paths and file names for source files
- Paths and file names for object files
- Compile flags
- Link flags

Choose an Approach for Specifying Build Information

The following table lists situations to help you decide when to use an interactive or programmatic approach to specifying build information:

Situation	Approach
Creating code replacement entries for the first time.	Code Replacement Tool.
You used the Code Replacement Tool to create the entries for which the build information applies.	Code Replacement Tool to specify the build information quickly .
Rapid prototyping entries.	Code Replacement Tool to generate, register, and test entries quickly.

Situation	Approach
Developing an entry to use as a template or starting point for defining similar entries.	Code Replacement Tool to generate entry code that you can copy and modify.
Modifying existing mappings.	MATLAB Editor to update the programming interface directly.

- If an entry uses header, source, or object files, consider whether to make the files accessible to the code generator. You can copy files to the build folder or you can specify individual file names and paths explicitly.
- If you specify *additional* header files/include paths or source files/paths and you copy files, the compiler and utilities such as `packNGo` might find duplicate instances of files (an instance in the build folder and an instance in the original folder).
- If you choose to copy files to the build folder and you are using the `packNGo` function to relocate static and generated code files to another development environment:
  - In the call to `packNGo`, specify the property-value pair `'minimalHeaders' true` (the default). That setting instructs the function to include the minimal header files required to build the code in the zip file.
  - Do not collocate files that you copy with files that you do not copy. If the `packNGo` function finds multiple instances of the same file, the function returns an error.
- If you use the programming interface, paths that you specify can include tokens. A token is a variable defined as a character vector or cell array of character vectors in the MATLAB workspace that you enclose with dollar signs (`$`*`variable`*`$`). The code generator evaluates and replaces a token with the defined value. For example, consider the path `$myfolder$\folder1`, where `myfolder` is a character vector variable defined in the MATLAB workspace as `'d:\work\source\module1'`. The code generator generates the custom path as `d:\work\source\module1\folder1`.

Specify Build Information Interactively with the Code Replacement Tool

The Code Replacement Tool provides a quick, easy way for you to specify build information for code replacement table entries. It is ideal for getting started with defining a table entry, rapid prototyping, and developing table entries to use as a starting point for defining similar mappings.

1 Determine the information that you must specify.

2. Open the Code Replacement Tool.
3. Select the code replacement table entry for which you want to specify the build information. In the left pane, select the table that contains the entry. In the middle pane, select the entry that you want to modify.
4. In the right pane, select the **Build Information** tab.
5. On the **Build Information** tab, specify your build information.

Parameter	Specify
Implementation header file	File name and extension for the header file the code generator needs to generate the replacement code. For example, `sin_dbl.h`.
Implementation source file	File name and extension for the C or C++ source file the code generator needs to generate the replacement code. For example, `sin_dbl.c`.
Additional header files/include paths	Paths and file names for additional header files the code generator needs to generate the replacement code. For example, `C:\libs\headerFiles` and `C:\libs\headerFiles\common.h`. This parameter adds `-I` to the compile line in the generated makefile.
Additional source files/ paths	Paths and file names for additional source files the code generator needs to generate the replacement code. For example, `C:\libs\srcFiles` and `C:\libs\srcFiles\common.c`. This parameter adds `-I` to the compile line in the generated makefile.
Additional object files/ paths	Paths and file names for additional object files the linker needs to build the replacement code. For example, `C:\libs\objFiles` and `C:\libs\objFiles\common.obj`.
Additional link flags	Flags the linker needs to generate an executable file for the replacement code.
Additional compile flags	Flags the compiler needs to generate object code for the replacement code.
Copy files to build directory	Whether to copy header, source, or object files, which are required to generate replacement

Parameter	Specify
	code, to the build folder before code generation. If you specify files with **Additional header files/include paths** or **Additional source files/ paths** and you copy files, the compiler and utilities such as `packNGo` might find duplicate instances of files.

6 Click **Apply**.

7 Select the **Mapping Information** tab. Scroll to the bottom of that table and click **Validate entry**. The tool validates the changes that you made to the entry.

8 Save the table that includes the entry that you just modified.

Specify Build Information Programmatically

The programming interface for specifying build information for a code replacement entry is ideal for:

- Modifying entries created with the Code Replacement Tool.
- Replicating and then modifying similar entries and tables.

The basic workflow for specifying build information programmatically is:

1 Identify or create the code replacement entry that you want to specify the build information.

2 Determine what information to specify.

3 Specify your build information.

Specify	Action

Implementation header file	Use one of the following:
	• Set properties `ImplementationHeaderFile` and `ImplementationHeaderPath` in a call to `setTflCFunctionEntryParameters`, `setTflCOperationEntryParameters`, or `setTflCSemaphoreEntryParameters`. For example:
	`setTflCFunctionEntryParameters(hEnt, ...`
	`'ImplementationHeaderFile', 'sin_dbl.h', ...`

Specify	Action

	`'ImplementationHeaderPath', 'D:/lib/headerFiles'`
	`'Key', 'sin', ...`
	`'ImplementationName', 'sin_dbl');`
	• Set argument `headerFile` in a call to `registerCFunctionEntry`, `registerCPPFunctionEntry`, or `registerCPromotableMacroEntry`

Implementation source file	Set properties `ImplementationSourceFile` and `ImplementationSourcePath` in a call to `setTflCFunctionEntryParameters`, `setTflCOperationEntryParameters`, or `setTflCSemaphoreEntryParameters`. For example:
	`setTflCFunctionEntryParameters(hEnt, ...`
	`'ImplementationHeaderFile', 'sin_dbl.c', ...`
	`'ImplementationHeaderPath', 'D:/lib/sourceFiles'`
	`'Key', 'sin', ...`
	`'ImplementationName', 'sin_dbl');`

Additional header files/include paths	For each file, specify the file name and path in calls to the functions `addAdditionalHeaderFile` and `addAdditionalIncludePath`. For example:
	`libdir = fullfile('$(MATLAB_ROOT)','..', '..', 'lib');`
	`hEnt = RTW.TflCFunctionEntry;`
	`addAdditionalHeaderFile(hEnt, 'common.h');`
	`addAdditionalIncludePath(hEnt, fullfile(libdir, 'include'));`
	These functions add `-I` to the compile line in the generated makefile.

Specify	Action

Additional source files/paths	For each file, specify the file name and path in calls to the functions `addAdditionalSourceFile` and `addAdditionalSourcePath`. For example:
	
	`libdir = fullfile('$(MATLAB_ROOT)','..', '..', 'lib');`
	
	`hEnt = RTW.TflCFunctionEntry;`
	
	`addAdditionalSourceFile(hEnt, 'common.c');`
	`addAdditionalSourcePath(hEnt, fullfile(libdir, 'src'));`
	
	These functions add `-I` to the compile line in the generated makefile.

Additional object files/paths	For each file, specify the file name and path in calls to the functions `addAdditionalLinkObj` and `addAdditionalLinkObjPath`. For example:
	
	`libdir = fullfile('$(MATLAB_ROOT)','..', '..', 'lib');`
	
	`hEnt = RTW.TflCFunctionEntry;`
	
	`addAdditionalLinkObj(hEnt, 'sin.o');`
	`addAdditionalLinkObjPath(hEnt, fullfile(libdir, 'bin'));`

Compile flags	Set the entry property `AdditionalCompileFlags` to a cell array of character vectors representing the required compile flags. For example:
	
	`hEnt = RTW.TflCFunctionEntry;`
	
	`hEnt.AdditionalCompileFlags = {'-Zi -Wall', '-O3'};`

Link flags	Set the entry property `AdditionalLinkFlags` to a cell array of character vectors representing the required link flags. For example:
	
	`hEnt = RTW.TflCFunctionEntry;`
	
	`hEnt.AdditionalCompileFlags = {'-MD -Gy', '-T'};`

Specify	Action

Whether to copy header, source, or object files, which are required to generate replacement code, to the build folder before code generation	Use one of the following:
	• Set property `GenCallback` to `'RTW.copyFileToBuildDir'` in a call to `setTflCFunctionEntryParameters`, `setTflCOperationEntryParameters`, or `setTflCSemaphoreEntryParameters`. For example:
	`setTflCFunctionEntryParameters(hEnt, ...`
	`'ImplementationHeaderFile', 'sin_dbl.h', ...`
	`'ImplementationHeaderPath', 'D:/lib/headerFiles'`
	`'Key', 'sin', ...`
	`'ImplementationName', 'sin_dbl'`
	`'GenCallback', 'RTW.copyFileToBuildDir');`
	• Set argument `genCallback` in a call to `registerCFunctionEntry`, `registerCPPFunctionEntry`, or `registerCPromotableMacroEntry` to `'RTW.copyFileToBuildDir'`.
	If a match occurs for a table entry, a call to the function `RTW.copyFileToBuildDir` copies required files to the build folder.
	If you specify additional header files/include paths or additional source files/paths and you copy files, the compiler and utilities such as `packNGo` might find duplicate instances of files.

4 Save the table that includes the entry that you added or modified.

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 replaced with calls to the optimized function. The optimized function does not reside in the build folder. For the code generator to access the files, copy them into the build folder to be compiled and linked into the application.

The table entry specifies the source and header file names and paths. To request the copy operation, the table entry sets the `genCallback` property to `'RTW.copyFileToBuildDir'` in the call to 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 a match occurs for the table

entry, the function `RTW.copyFileToBuildDir` copies the specified source and header files to the build folder for use during the remainder of the build process.

```
function hTable = make_my_crl_table

hTable = RTW.TflTable;

op_entry = RTW.TflCOperationEntry;
setTflCOperationEntryParameters(op_entry, ...
                'Key',                      'RTW_OP_MUL', ...
                'Priority',                 100, ...
                'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
                'RoundingModes',            {'RTW_ROUND_UNSPECIFIED'}, ...
                'ImplementationName',       's32_mul_s32_s32_sat', ...
                'ImplementationHeaderFile', 's32_mul.h', ...
                'ImplementationSourceFile', 's32_mul.s', ...
                'ImplementationHeaderPath', {fullfile('$(MATLAB_ROOT)','crl')}, ...
                'ImplementationSourcePath', {fullfile('$(MATLAB_ROOT)','crl')}, ...
                'GenCallback',              'RTW.copyFileToBuildDir');
.
.
.
addEntry(hTable, op_entry);
```

The following example uses the functions `addAdditionalHeaderFile`, `addAdditionalIncludePath`, `addAdditionalSourceFile`, `addAdditionalSourcePath`, `addAdditionalLinkObj`, and `addAdditionalLinkObjPath` in addition to the code generation callback function `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', ...
                'RoundingModes',            {'RTW_ROUND_UNSPECIFIED'}, ...
                'ImplementationName',       's32_add_s32_s32', ...
                'ImplementationHeaderFile', 's32_add_s32_s32.h', ...
                'ImplementationSourceFile', 's32_add_s32_s32.c'...
                'GenCallback',              'RTW.copyFileToBuildDir');

addAdditionalHeaderFile(op_entry, 'all_additions.h');
addAdditionalIncludePath(op_entry, fullfile(libdir, 'include'));
addAdditionalSourceFile(op_entry, 'all_additions.c');
addAdditionalSourcePath(op_entry, fullfile(libdir, 'src'));
addAdditionalLinkObj(op_entry, 'addition.o');
addAdditionalLinkObjPath(op_entry, fullfile(libdir, 'bin'));
.
```

```
.
.
addEntry(hTable, op_entry);
```

Next, include your code replacement table in a code replacement library and register the library with the code generator.

More About

- “Code You Can Replace From Simulink Models” on page 51-7
- “Define Code Replacement Mappings” on page 51-42
- “Register Code Replacement Mappings” on page 51-68
- “Develop a Code Replacement Library” on page 51-27
- “What Is Code Replacement Customization?” on page 51-3

Register Code Replacement Mappings

After you define code replacement entries and specify build information in a code replacement table, you can include the table in a code replacement library that you register with the code generator. When registered, a library appears in the list of available code replacement libraries that you can choose from when configuring the code generator.

Register a code replacement table as a code replacement library:

- Interactively, by using the Code Replacement Tool
- Programmatically, by using a MATLAB programming interface

Choose an Approach for Creating the Registration File

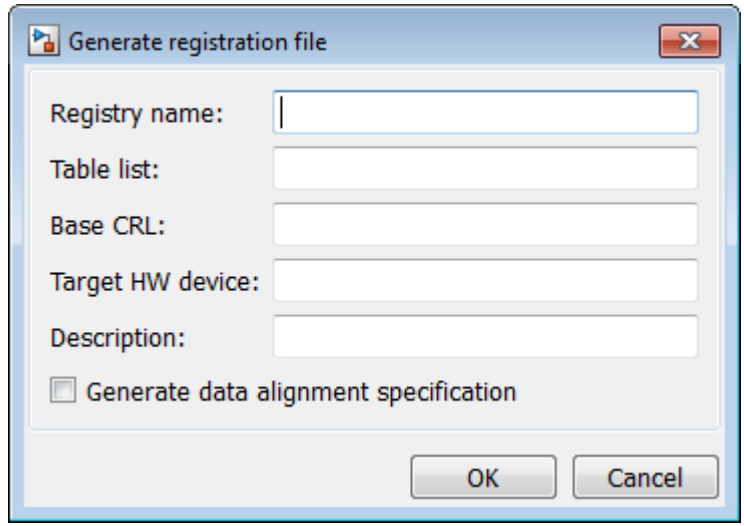
The following table lists situations to help you decide when to use an interactive or programmatic approach to creating a registration file:

If...	Then...
Registering a code replacement table for the first time	Use the Code Replacement Tool.
You used the Code Replacement Tool to create the table	Use the Code Replacement Tool to quickly register the table.
Rapid prototyping code replacement	Use the Code Replacement Tool to quickly generate, register, and test entries.
Creating registration file to use as a template or starting point for defining similar registration files	Use the Code Replacement Tool to generate code that you can copy and modify.
Modifying existing registration files	Use the MATLAB Editor to update the registration file.
Defining multiple code replacement libraries in one registration file	Use the MATLAB Editor to create a new or extend an existing registration file.
Defining code replacement library hierarchy in a registration file	Use the MATLAB Editor to create a new or extend an existing registration file.

Create Registration File Interactively with the Code Replacement Tool

The Code Replacement tool provides a quick, easy way for you to create a registration file for a code replacement table. It is ideal for getting started, rapid prototyping, and generating a registration file that you want to use as a starting point for similar registrations.

1 After you validate and save a code replacement table, select **File > Generate registration file** to open the **Generate registration file** dialog box.



2 Enter the registration information. Minimally, specify:

For...	Specify...
Registry name	Text naming the code replacement library. For example, `Sin Function Example`.
Table list	Text naming one or more code replacement tables to include in the library. Specify each table as one of the following: • Name of a table file on the MATLAB search path • Absolute path to a table file • Path to a table file relative to `$(MATLAB_ROOT)`

For...	Specify...
	You can specify multiple tables. If you do, separate the table specifications with a comma. For example: `crl_table_sinfcn, c:/work_crl/crl_table_muldiv` See "Registration Files That Define Multiple Code Replacement Libraries" on page 52-61 for examples of each type of table specification.

Optionally, you can specify:

For...	Specify...
Description	Text that describes the purpose and content of the library.
Target HW device	Text naming one or more hardware devices the code replacement library supports. Separate names with a comma. To support all device types, enter an asterisk (*). For example, `TI C28x, TI C62x`.
Base CRL	Text naming a code replacement library that you want to serve as a base library for the library you are registering. Use this field to specify library hierarchies. For example, you can specify a general `TI device library` as the base library for a more specific `TI C28x` device library.
Generate data alignment specification	Flag that enables data alignment specification.

Create Registration File Programmatically

The programming interface for creating a registration file for a code replacement table is ideal for:

- Modifying registration files created with the Code Replacement Tool
- Replicating and modifying similar registration files
- Defining multiple code replacement libraries in one registration file

The basic workflow for creating a registration file programmatically consists of the following steps:

1. Define an `rtwTargetInfo` function. The code generator recognizes this function as a customization file. The function definition must include at least the following content:

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@loc_register_crl);

function this = loc_register_crl

this(1) = RTW.TflRegistry;
this(1).Name = 'crl-name';
this(1).TableList = {'table',...};
```

For...	Replace...
`this(1).Name = 'crl-name';`	*`crl-name`* with text naming the code replacement library. For example, `Sin Function Example`.
`this(1).TableList = {'table',...};`	*`table`* with text that identifies the code replacement table that contains your code replacement entries. Specify a table as one of the following: • Name of a table file on the MATLAB search path • Absolute path to a table file • Path to a table file relative to `$(MATLAB_ROOT)` You can specify multiple tables. If you do, separate the table specifications with commas.

Optionally, you can specify:

For...	Replace...
`this(1).Description = '`*`text`*`'`	*`text`* with text that describes the purpose and content of the library.
`this(1).TargetHWDeviceType = {'`*`device-type`*`',...}`	*`device-type`* with text that names a hardware device the code replacement library supports. You can specify multiple device types. Separate device types with a comma. For example, `TI C28x, TI C62x`. To support all device types, enter an asterisk (*).
`this(1).BaseTfl = '`*`base-lib`*`'`	*`base-lib`* with text that names a code replacement library that you want to serve as a base library for the library you are registering. Use this field to specify library hierarchies. For example, you can specify a general `TI device` library as the base library for a `TI C28x device` library. See "Registration Files That Define Code Replacement Library Hierarchies" on page 52-61 for an example.

For example:

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@loc_register_crl);

function this = loc_register_crl

this(1) = RTW.TflRegistry;
this(1).Name = 'Sin Function Example';
this(1).TableList = {'crl_table_sinfcn'};
this(1).TargetHWDeviceType = {'*'};
this(1).Description = 'Example - sin function replacement';
```

2 Save the file with the name `rtwTargetInfo.m`.

3 Place the file on the MATLAB path. When the file is on the MATLAB path, the code generator reads the file after starting and applies the customizations during the current MATLAB session.

Register a Code Replacement Library

Before you can use the code replacement tables defined in a registration file, refresh Simulink customizations within the current MATLAB session. To initiate a refresh, enter the following command:

```
sl_refresh_customizations
```

Register a Library that Includes Multiple Code Replacement Tables

Use the programming interface to create a registration file that defines a code replacement library that includes multiple code replacement tables. The following example defines a library that includes multiple tables. The `TableList` fields specify tables that reside at different locations. The tables reside on the MATLAB search path or at locations specified with a path.

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@locCrlRegFcn);

function thisCrl = locCrlRegFcn

  % Register a code replacement library for use with model: rtwdemo_crladdsub
  thisCrl(1) = RTW.TflRegistry;
  thisCrl(1).Name = 'Addition & Subtraction Examples';
  thisCrl(1).Description = 'Example of addition/subtraction op replacement';
  thisCrl(1).TableList = {'crl_table_addsub'};
  thisCrl(1).TargetHWDeviceType = {'*'};

  % Register a code replacement library for use with model: rtwdemo_crlmuldiv
  thisCrl(2) = RTW.TflRegistry;
  thisCrl(2).Name = 'Multiplication & Division Examples';
  thisCrl(2).Description = 'Example of mult/div op repl for built-in integers';
  thisCrl(2).TableList = {'c:/work_crl/crl_table_muldiv'};
  thisCrl(2).TargetHWDeviceType = {'*'};

  % Register a code replacement library for use with model: rtwdemo_crlfixpt
  thisCrl(3) = RTW.TflRegistry;
  thisCrl(3).Name = 'Fixed-Point Examples';
  thisCrl(3).Description = 'Example of fixed-point operator replacement';
  thisCrl(3).TableList = {fullfile('$(MATLAB_ROOT)', ...
     'toolbox','rtw','rtwdemos','crl_demo','crl_table_fixpt')};
  thisCrl(3).TargetHWDeviceType = {'*'};
```

Registration Files That Define Code Replacement Library Hierarchies

Using the programming interface, you can organize multiple code replacement libraries in a hierarchy. The following example shows a registration file that defines four code

replacement tables organized in a hierarchy of four code replacement libraries. The tables include entries that increase in specificity: common entries, entries for TI devices, entries for TI C6xx devices, and entries specific to the TI C67x device.

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@locCrlRegFcn);

function thisCrl = locCrlRegFcn

  % Register a code replacement library that includes common entries
  thisCrl(1) = RTW.TflRegistry;
  thisCrl(1).Name = 'Common Replacements';
  thisCrl(1).Description = 'Common code replacement entries shared by other libraries';
  thisCrl(1).TableList = {'crl_table_general'};
  thisCrl(1).TargetHWDeviceType = {'*'};

  % Register a code replacement library for TI devices
  thisCrl(2) = RTW.TflRegistry;
  thisCrl(2).Name = 'TI Device Replacements';
  thisCrl(2).Description = 'Code replacement entries shared across TI devices';
  thisCrl(2).TableList = {'crl_table_TI_devices'};
  thisCrl(2).TargetHWDeviceType = {'TI C28x', 'TI C55x', 'TI C62x', 'TI C64x', 'TI 67x'};
  thisCrl(1).BaseTfl = 'Common Replacements'

  % Register a code replacement library for TI c6xx devices
  thisCrl(3) = RTW.TflRegistry;
  thisCrl(3).Name = 'TI c6xx Device Replacements';
  thisCrl(3).Description = 'Code replacement entries shared across TI C6xx devices';
  thisCrl(3).TableList = {'crl_table_TIC6xx_devices'};
  thisCrl(3).TargetHWDeviceType = {'TI C62x', 'TI C64x', 'TI 67x'};

  % Register a code replacement library for the TI c67x device
  thisCrl(3) = RTW.TflRegistry;
  thisCrl(3).Name = 'TI c67x Device Replacements';
  thisCrl(3).Description = 'Code replacement entries for the TI C67x device';
  thisCrl(3).TableList = {'crl_table_TIC67x_device'};
  thisCrl(3).TargetHWDeviceType = {'TI 67x'};
```

After registering your code replacement mappings, verify that code replacements occur.

More About

- “Troubleshoot Code Replacement Library Registration” on page 51-75
- “Specify Build Information for Replacement Code” on page 51-59
- “Verify Code Replacements” on page 51-76
- “Develop a Code Replacement Library” on page 51-27
- “What Is Code Replacement Customization?” on page 51-3

Troubleshoot Code Replacement Library Registration

If a code replacement library is not listed as a configuration option or does not appear in the Code Replacement Viewer:

- Refresh the library registration information within the current MATLAB session (`RTW.TargetRegistry.getInstance('reset');` or for the Simulink environment,`sl_refresh_customizations`).
- See whether the registration file, `rtwTargetInfo.m`, contains an error.

More About

- "Register Code Replacement Mappings" on page 51-68

Verify Code Replacements

After you create or modify and register a code replacement table, use the following techniques to examine and verify the table and its entries.

- Invoke the table definition file at the command prompt.
- Use the Code Replacement Viewer to examine libraries, tables, and entries.
- Trace code replacements from the source where you applied the code replacement library.
- Examine code replacement hits and misses logged during code generation.

Code Replacement Hits and Misses

The code generator logs code replacement table entries for which it finds and does not find matches in the hit cache and miss cache, respectively. When a code replacement entry match fails and code is not replaced, the code generator logs the call site object (CSO) for the miss in the miss cache. When an entry match succeeds, the code generator logs the matched entry in the hit cache.

The code generator overwrites the hit and miss cache data each time it produces code. The cache data reflects hits and misses for only the last application component (MATLAB code or Simulink model) for which you generate code.

You can use the Code Replacement Viewer to review trace information based on logged hit and miss trace data. The hit cache provides trace information that helps to verify code replacements.

The miss cache and related miss data collected and stored in code replacement tables provide trace information for misses. Use this information for misses to troubleshoot expected code replacements that do not occur. Trace information for a miss:

- Identifies the call site object.
- Provides a link to the relevant source location for the miss.
- Includes information about the reason for the miss.

Validate Table Definition File

After you create or modify a code replacement table definition file, validate it. At the command prompt, specify the name of the table in a call to the `isvalid` function. For example:

```
isvalid(crl_table_sinfcn)

ans =

      1
```

MATLAB displays errors that occur. In the following example, MATLAB detects a typo in a data type name.

```
isvalid(crl_table_sinfcn)

??? RTW_CORE:tfl:TflTable: Unsupported data type, 'dooble'.

Error in ==> crl_table_sinfcn at 7
hTable.registerCFunctionEntry(100, 1, 'sin', 'dooble', 'sin_dbl', ...
```

Review Library Content

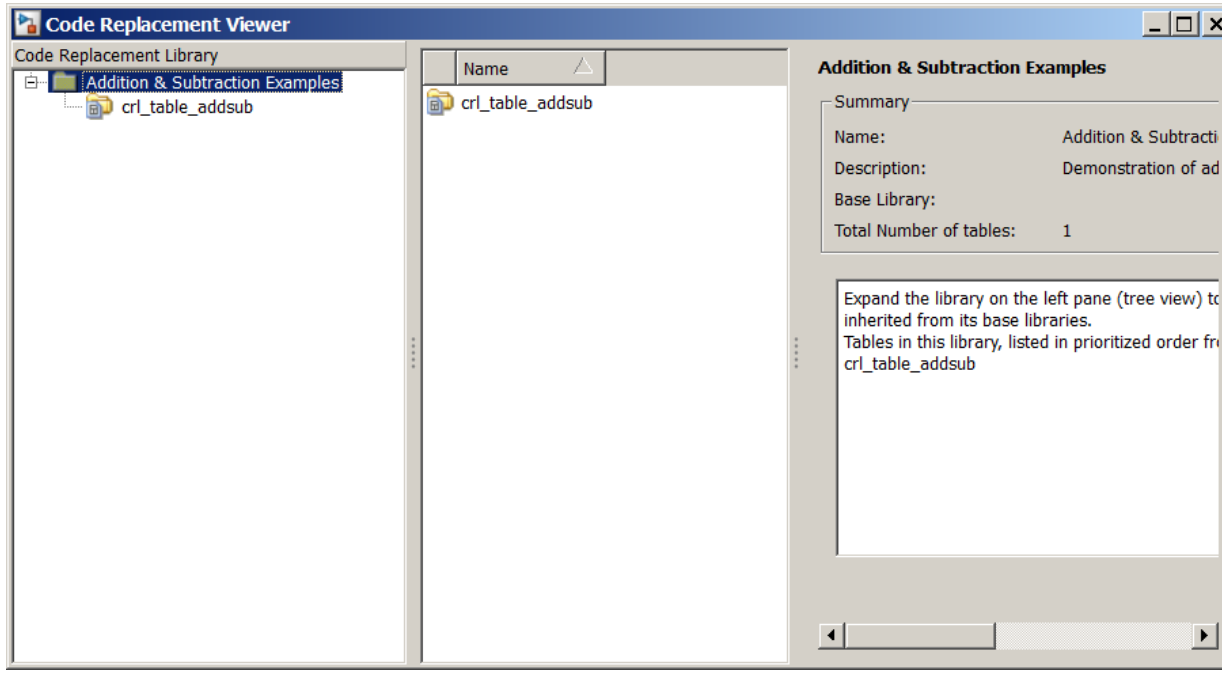
After you create or modify a code replacement library, use the Code Replacement Viewer to review and verify the list of tables in the library and the entries in each table.

1 Open the viewer to display the contents of your library. At the command prompt, enter the following command:

```
crviewer('library')
```

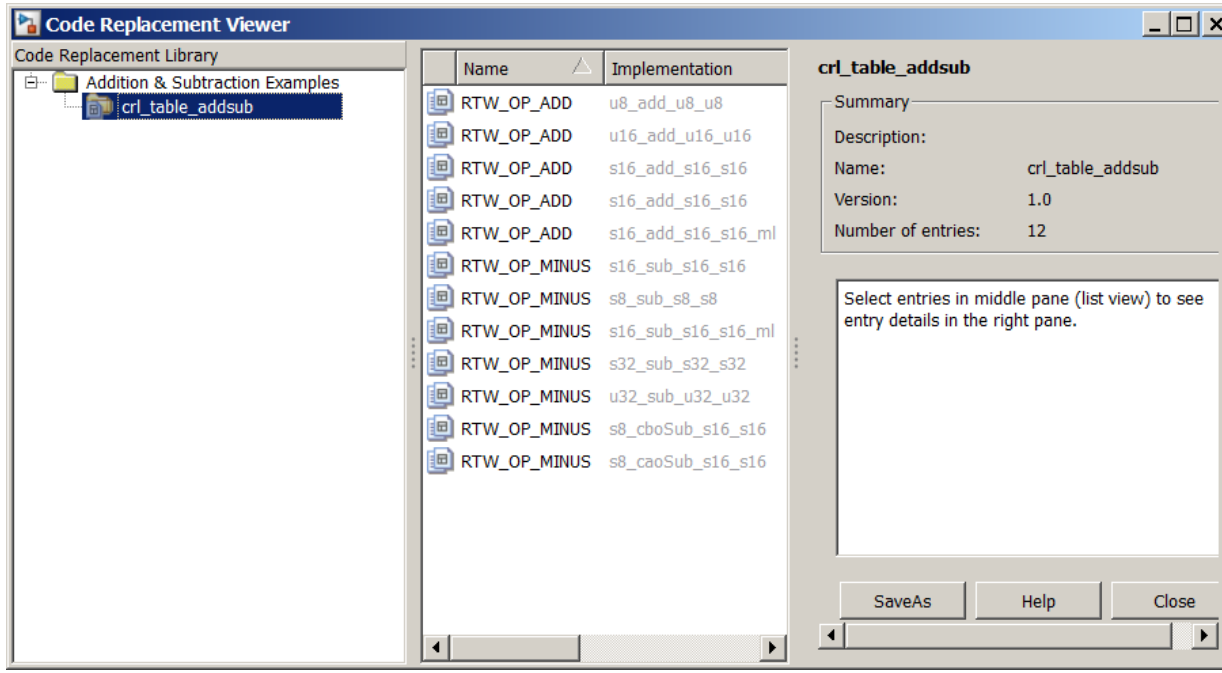
For example:

```
crviewer('Addition & Subtraction Examples')
```



2 Review the list of tables in the left pane. Are tables missing? Are the tables listed in the correct relative order? By default, the viewer displays tables in search order.

3 In the left pane, click each table and review the list of entries in the center pane. Are entries missing? Does the list include extraneous or unexpected entries?



Review Table Content

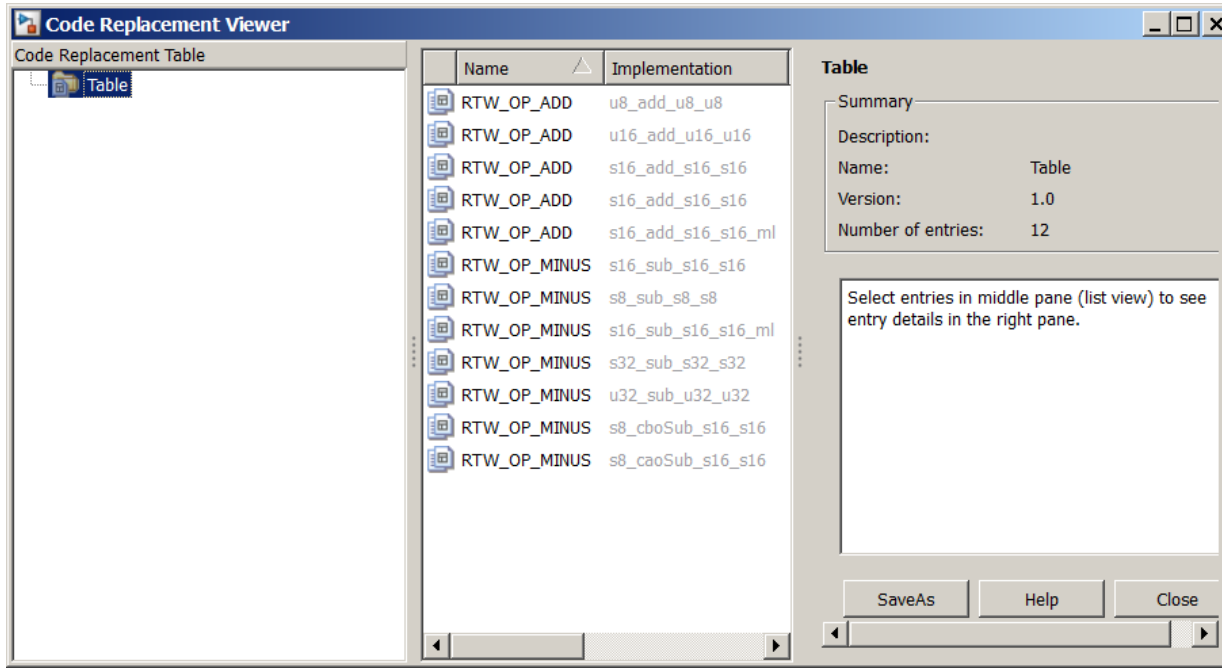
After you create or modify a code replacement table, use the Code Replacement Viewer to review and verify table entries.

1 Open the viewer to display the contents of your table. At the command prompt, enter the following command. *`table`* is a MATLAB file that defines code replacement tables. The file must be in the current folder or on the MATLAB path.

```
crviewer(table)
```

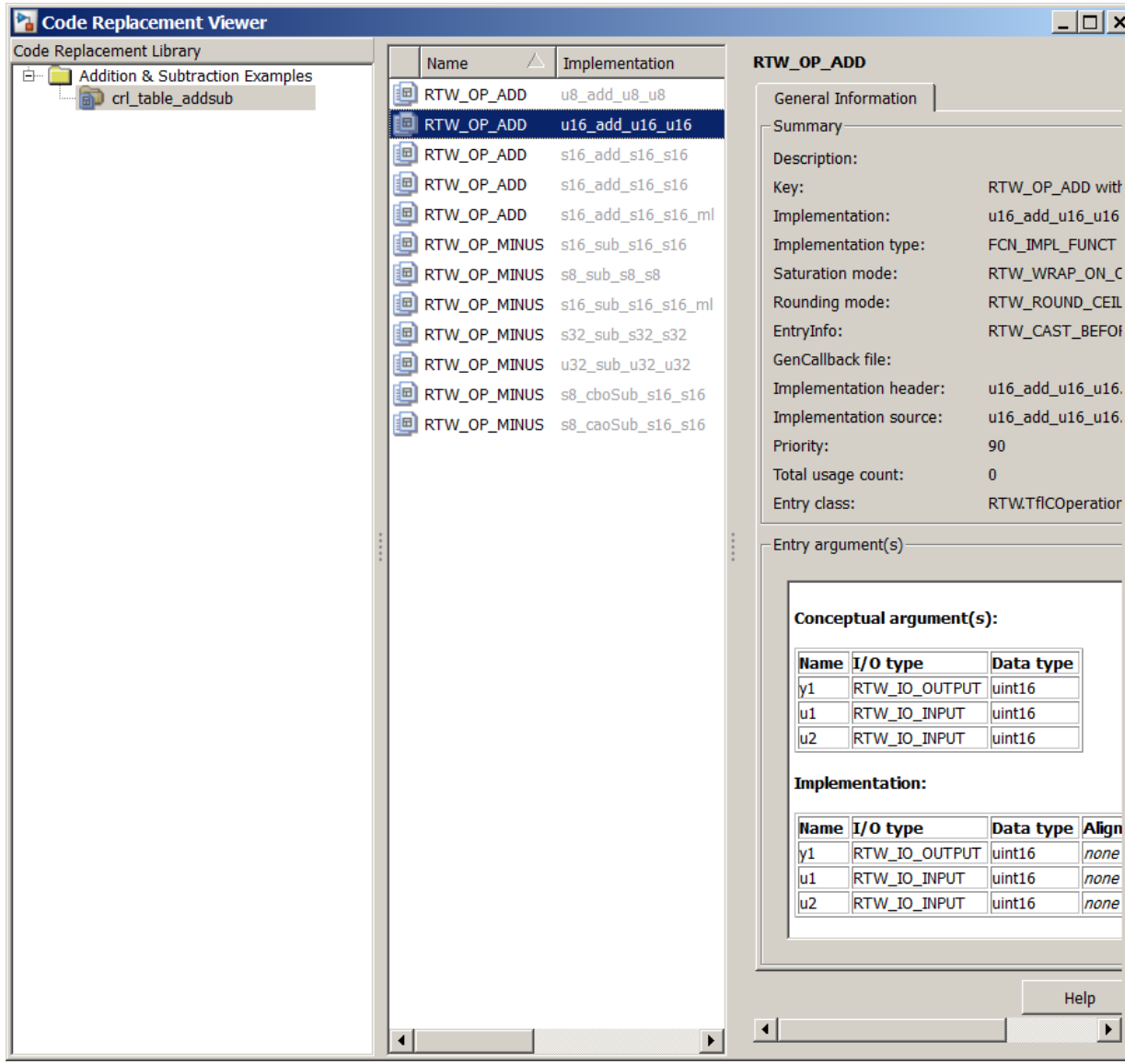
For example:

```
crviewer(crl_table_addsub)
```



2 Review the list of entries in the center pane. Are entries missing? Does the list include extraneous or unexpected entries? By default, the viewer displays entries in search order.

3 In the center pane, click each entry and verify the entry information in the right pane.



- Argument order is correct.

- Conceptual argument names match code generator naming conventions.
- Implementation argument names are correct.
- Algorithm properties (for example, saturation and rounding mode) are set correctly.
- Header or source file specification is not missing.
- I/O types are correct.
- Relative priority of entries is correct.

Review Code Replacements

After you review the content of your code replacement library and tables, generate code and a code generation report. Verify that the code generator replaces code as you expect.

The Code Replacements Report details the code replacement library functions that the code generator uses for code replacements. The report provides a mapping between each replacement instance and the model element that triggered the replacement.

The following example illustrates two complementary approaches to reviewing code replacements:

- Check the Code Replacements Report section of the code generation report for expected replacements.
- Trace code replacements.

For models that consist of model hierarchies, repeat the following procedure for each model in the hierarchy. Generate code for and review the trace information of each referenced model separately. Logged cache hit and miss information captured in the Code Replacement Viewer is valid for the last model for which code was generated. As you generate code for each model in the hierarchy, the code generator overwrites logged information.

1. Open the model where you anticipate that a function or operator replacement occurs. This example uses the model `rtwdemo_crladdsub`.
2. Configure the code generator to use your code replacement library. For this example, set the library to `Addition & Subtraction Examples`.
3. Configure the code generation report to include the Code Replacements Report. On the **Code Generation** > **Report** pane, select **Create code generation report** and

Open report automatically. On the **All Parameters** tab, select **Model-to-code** and **Summarize which blocks triggered code replacements**.

4 Configure comments for the generated code. On the **Code Generation** > **Comments** pane, select:

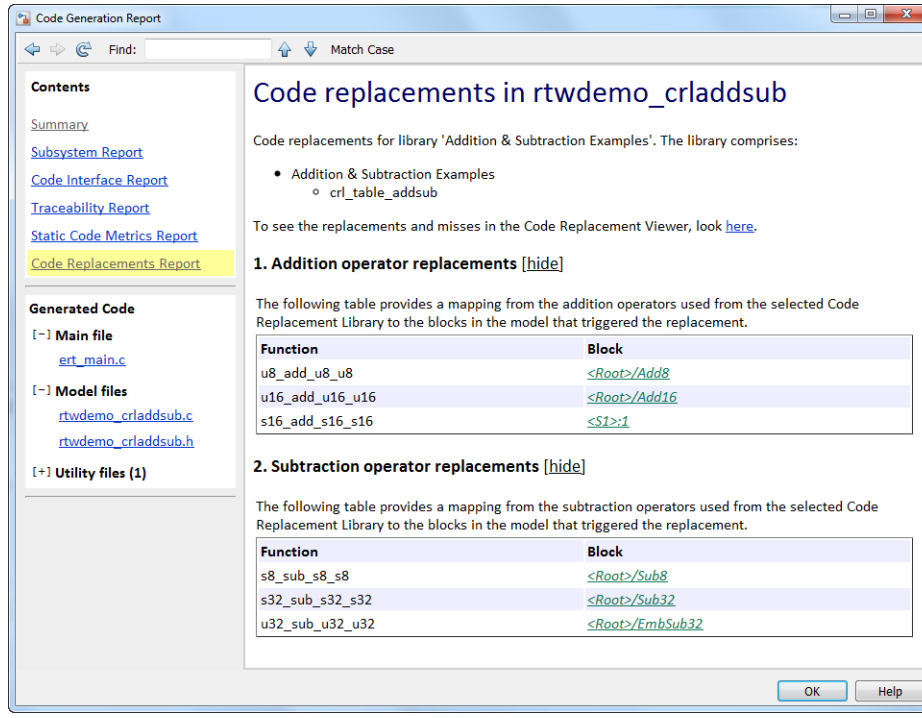
- **Include comments**
- Either or both of **Simulink block / Stateflow object comments** and **Simulink block descriptions**

In the **Code Replacements Report**, these options include Simulink block information.

5 Configure the code generator to generate only code. Before you build an executable file, review your code replacements in the generated code.

6 Generate code and a report.

7 Open the **Code Replacements Report** section of the code generation report.



The report lists the replacement functions that the code generator used. It provides a mapping between each replacement instance and the Simulink block that triggered the replacement.

Review the report:

- Check whether expected function and operator code replacements occurred.
- In the replacements sections, click each block link to see the source that triggered the reported code replacement.

8 In the Simulink model window, use model-to-code highlighting to trace code replacements. Identify and right-click a block where you expected code replacement to occur. Select **C/C++ Code** > **Navigate to C/C++ Code**. The code generation report appears with the corresponding replacement code highlighted. In the example model rtwdemo_crladdsub, right-click the Add8 block and select **C/C++ Code** > **Navigate to C/C++ Code**.

```
24 /* Real-time model */
25 RT_MODEL rtM_;
26 RT_MODEL *const rtM = &rtM_;
27
28 /* Model step function */
29 void rtwdemo_crladdsub_step(void)
30 {
31   /* Outport: '<Root>/Out1' incorporates:
32    *  Inport: '<Root>/In1'
33    *  Inport: '<Root>/In2'
34    *  Sum: '<Root>/Add8'
35    */
36   rtY.Out1 = u8_add_u8_u8(rtU.In1, rtU.In2);
37
38   /* Outport: '<Root>/Out2' incorporates:
39    *  Inport: '<Root>/In3'
40    *  Inport: '<Root>/In4'
41    *  Sum: '<Root>/Add16'
42    */
43   rtY.Out2 = u16_add_u16_u16(rtU.In3, rtU.In4);
```

Inspect the generated code to see if the function or operator replacement occurred as you expected.

If a function or operator is not replaced as expected, the code generator used a higher-priority (lower-priority value) match or did not find a match.

To analyze and troubleshoot code replacement misses, use the trace information that the Code Replacement Viewer provides. See “Troubleshoot Code Replacement Misses” on page 51-86.

Next, deploy your code replacement library for others to use.

More About

- “Troubleshoot Code Replacement Misses” on page 51-86
- “Register Code Replacement Mappings” on page 51-68
- “Deploy Code Replacement Library” on page 51-93
- “What Is Code Replacement Customization?” on page 51-3

Troubleshoot Code Replacement Misses

Use miss reason messages that appear in the Code Replacement Viewer to analyze and correct code replacement misses.

Miss Reason Messages

The Code Replacement Viewer displays miss reason messages in trace information for code replacement misses. A legend listing each message that appears in the miss report precedes the report details. A message consists of:

- Numeric identifier, which identifies the message in the report details.
- Message text, which in some cases includes placeholders for names of arguments, call site object values, table entry values, and property names.

For example:

```
1. Mismatched data types (argument name, CSO value, table entry value)
```

The parenthetical information represents placeholders for actual values that appear in the report details.

In the **Miss Source Locations** table that lists the miss details, the **Reason** column includes:

- The message identifier, as listed in the legend.
- The placeholder values for that instance of the miss reason message.

The following **Reason** details indicate a data type mismatch because the call site object specifies data type `int8` for arguments `y1`, `u1`, and `u2`, while the code replacement table entry specifies `uint32`.

```
1. y1, int8, uint32
   u1, int8, uint32
   u2, int8, uint32
```

Depending on your situation and the reported miss reason, troubleshoot reported misses by looking for instances of the following:

- A typo in the code replacement table entry definition or a source parameter setting.
- Information missing from the code replacement table entry or a source parameter setting.

- Invalid or incorrect information in the code replacement table entry definition or a source parameter setting.
- Arguments incorrectly ordered in the code replacement table entry definition or the source being replaced with replacement code.
- Failed algorithm classification for an addition or subtraction operation due to:
  - An ideal accumulator not being calculated because the type of an input argument is not fixed-point or the slope adjustment factors of the input arguments are not equal.
  - Input or output casts with a floating-point cast type.
  - Input or output casts with cast types that have different slope adjustment factors or biases.
  - Output casts not being convertible to a single output cast.
  - Input casts resulting in loss of bits.

Analyze and Correct Code Replacement Misses

The following example shows how to use Code Replacement Viewer trace information to troubleshoot code replacement misses. You must have already reviewed and tested code replacements for your model.

1 Review the code generated for a model element, looking for expected code replacements. For this example, examine the code generated for block Sub32 in model `rtwdemo_crladdsub`. Right-click the block and select **C/C++ Code** > **Navigate to C/C++ Code**.

The Code Generation Report opens to the location of the generated code for that block.

```
63   /* Outport: '<Root>/Out5' incorporates:
64    *  Inport: '<Root>/In10'
65    *  Inport: '<Root>/In9'
66    *  Sum: '<Root>/Sub32'
67    */
68   rtY.Out5 = s32_sub_s32_s32(rtU.In9, rtU.In10);
```

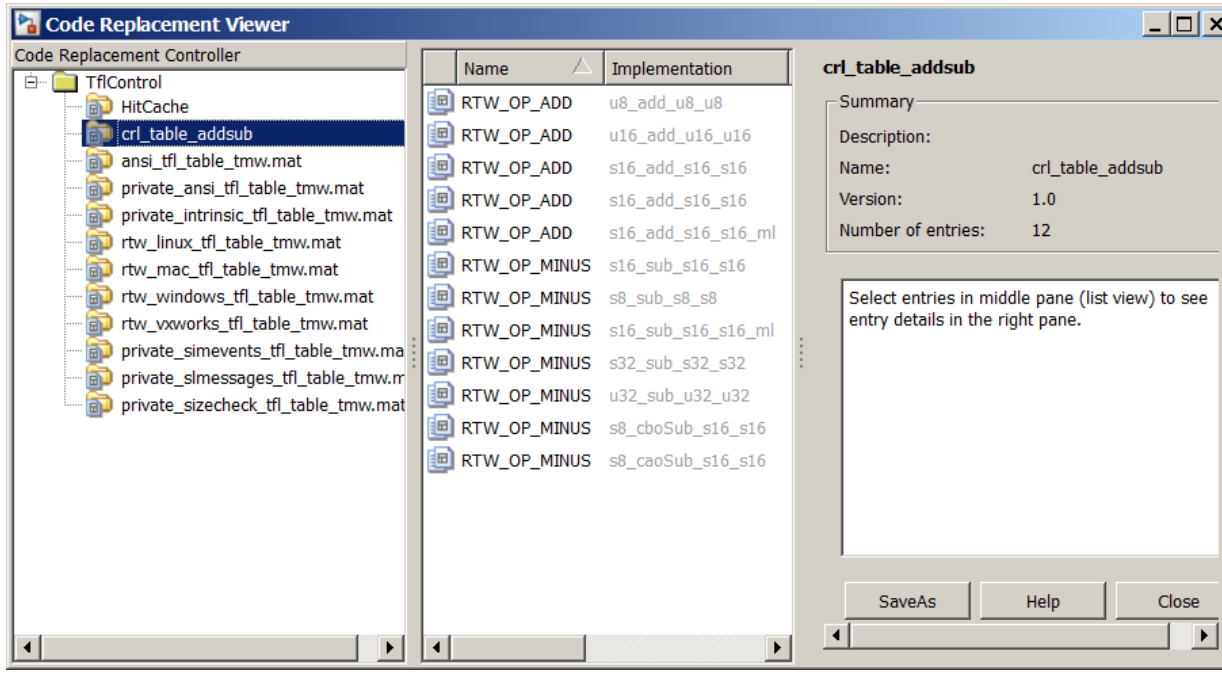
The code generator replaced code, but the replacement was for the signed version of the 32-bit subtraction operation. You expected an unsigned operation.

2 Regenerate or reopen the Code Replacements Report for your model. If you already generated the code generation report that includes the Code Replacements Report

for model rtwdemo_crladdsub, open the file `rtwdemo_crladdsub_ert_rtw/html/rtwdemo_crladdsub_codegen_rpt.html`. For information on how to regenerate the report, see "Review Code Replacements" on page 51-82.

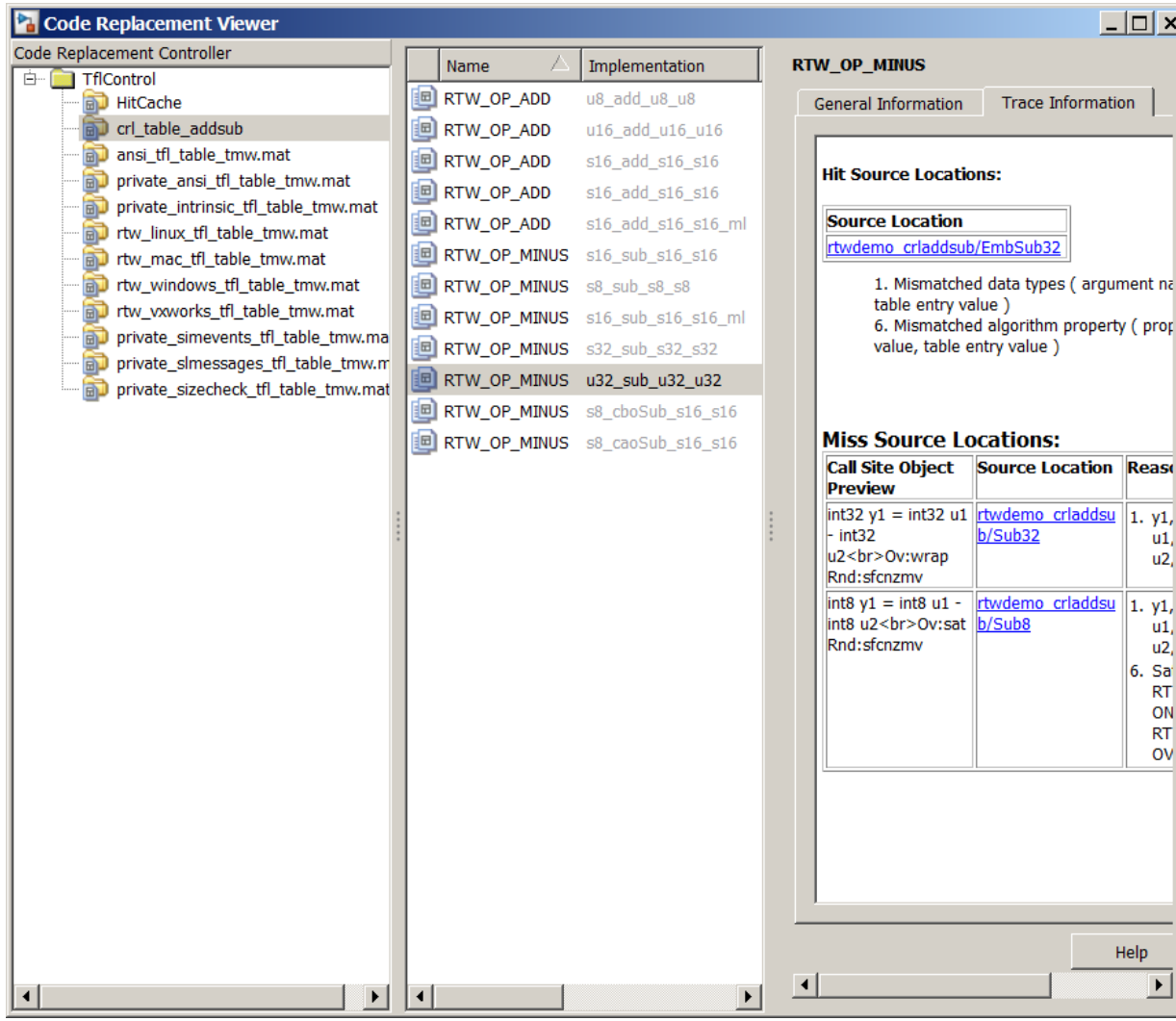
3 Click the link to open the Code Replacement Viewer.

4 In the viewer left pane, select your code replacement table. The following display shows entries for code replacement table `crl_table_addsub`.



5 In the middle pane, select table entry `RTW_OP_MINUS` with implementation function `u32_sub_u32_u32`.

6 In the right pane, select the **Trace Information** tab.



The **Trace Information** is a table that lists the following information for each miss:

- Call site object preview. The call site object is the conceptual representation of a subtraction operator. The code generator uses this object to query the code replacement library for a match.

- A link to the source location in the model for which the code generator considered replacing code.
- The reasons that the miss occurred. For the list of reasons that misses occur, see "Miss Reason Messages" on page 51-86.

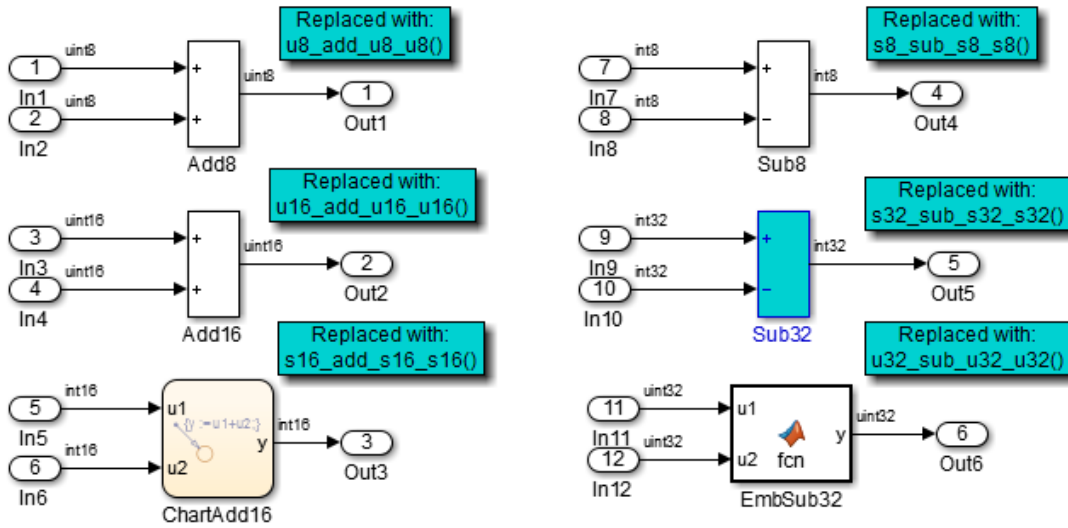
For this example, the report shows misses for two blocks: Sub32 and Sub8.

7 Find that source in the trace information. Depending on your situation and the reported miss reason, consider looking for a condition such as a typo in the code replacement table entry definition or in a source parameter setting. "Miss Reason Messages" on page 51-86 lists conditions to consider.

For this example, determine why code for the Sub32 block was not replaced with code for an unsigned 32-bit subtraction operation. The miss reason for the Sub32 block indicates a data type mismatch. The data type in the call site object for the three arguments is a signed 32-bit integer. The code replacement entry specifies an unsigned 32-bit integer.

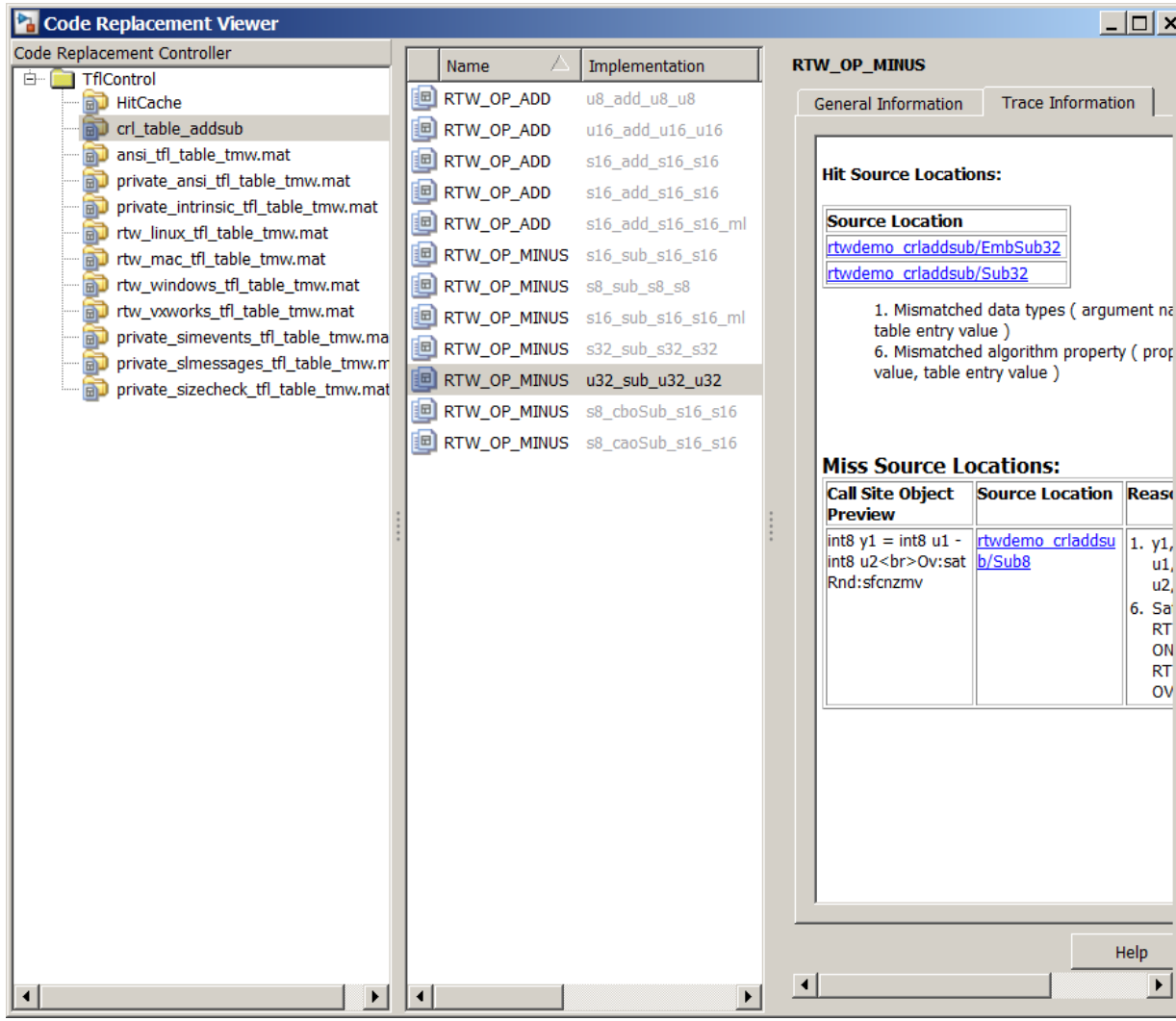
8 Correct the model or code replacement table entry. If the issue is in the model, use the source location link in the trace information to find the model element to correct. For this example, you expected an unsigned subtraction operation for the Sub32 block. Click the link in the trace report for the Sub32 block.

The model opens with the Sub32 block highlighted.



Change the data type setting for the two input signals and the output signal for the Sub32 block to `uint32`.

9 Regenerate code. Use the Code Replacement Viewer trace information to verify that your model or code replacement table entry corrects the code replacement issue. In the following display, the trace information shows a hit for block Sub32.



More About

- “Verify Code Replacements” on page 51-76

Deploy Code Replacement Library

After you verify code replacements and are ready to package and deploy a code replacement library for others to use:

1. Move your code replacement table files to an area that is on the MATLAB search path and that is accessible to and shared by other users.
2. Move the `rtwTargetInfo.m` registration file, to an area that is on the MATLAB search path and that is accessible to and shared by other users. If you are deploying a library to a folder in a development environment that already contains a `rtwTargetInfo.m` file, copy the registration code from your code replacement library version of `rtwTargetInfo.m` and paste it into the shared version of that file.
3. Register the library customizations or restart MATLAB.
4. Verify that the libraries are available for configuring the code generator and that code replacements occur as expected.
5. Inform users that the libraries are available and provide direction on when and how to apply them.

More About

- "Verify Code Replacements" on page 51-76
- "Relocate Code to Another Development Environment" (Simulink Coder)
- "Develop a Code Replacement Library" on page 51-27
- "What Is Code Replacement Customization?" on page 51-3

Math Function Code Replacement

This example shows how to define a code replacement mapping for a math function. The example defines a mapping for the `sin` function programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

1. Create a table definition file that contains a function definition. For example:

```
function hTable = crl_table_sinfcn2()
%CRL_TABLE_SINFCN2 - Define function entry for code replacement table.
```

2. Within the function body, create the table by calling the function `RTW.TflTable`.

```
hTable = RTW.TflTable;
```

3. Create an entry for the function mapping with a call to the `RTW.TflCFunctionEntry` function.

```
% Create entry for sin function replacement
fcn_entry = RTW.TflCFunctionEntry;
```

4. Set function entry parameters with a call to the `setTflCFunctionEntryParameters` function.

```
setTflCFunctionEntryParameters(fcn_entry, ...
                               'Key',                    'sin', ...
                               'Priority',               30, ...
                               'ImplementationName',     'mySin', ...
                               'ImplementationHeaderFile', 'basicMath.h',...
                               'ImplementationSourceFile', 'basicMath.c');
```

5. Create conceptual arguments `y1` and `u1`. There are multiple ways to set up the conceptual arguments. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call.

```
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');
```

6. Copy the conceptual arguments to the implementation arguments. There are multiple ways to set up the implementation arguments. This example uses a call to the `copyConceptualArgsToImplementation` function to create and add implementation arguments to the entry by copying matching conceptual arguments.

```
copyConceptualArgsToImplementation(fcn_entry);
```

7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, fcn_entry);
```

8 Save the table definition file. Use the name of the table definition function to name the file.

More About

- "Code You Can Replace From Simulink Models" on page 51-7
- "Define Code Replacement Mappings" on page 51-42
- "Algorithm-Based Code Replacement" on page 51-109
- "Data Alignment for Code Replacement" on page 51-133
- "Reserved Identifiers and Code Replacement" on page 51-152
- "Customize Match and Replacement Process" on page 51-153
- "Develop a Code Replacement Library" on page 51-27

Memory Function Code Replacement

This example shows how to define a code replacement mapping for a memory function. The example defines a mapping for the `memcpy` function programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

1. Create a table definition file that contains a function definition. For example:

   ```
   function hTable = crl_table_memcpy()
   ```

2. Within the function body, create the table by calling the function `RTW.TflTable`.

   ```
   hTable = RTW.TflTable;
   ```

3. Create an entry for the function mapping with a call to the `RTW.TflCFunctionEntry` function.

   ```
   % Create entry for void* memcpy(void*, void*, size_t)
   fcn_entry = RTW.TflCFunctionEntry;
   ```

4. Set function entry parameters with a call to the `setTflCFunctionEntryParameters` function.

   ```
   % Set SideEffects to 'true' for function returning void to prevent it from
   % being optimized away.
   setTflCFunctionEntryParameters(fcn_entry, ...
                                  'Key',                    'memcpy', ...
                                  'Priority',               90, ...
                                  'ImplementationName',     'memcpy_int', ...
                                  'ImplementationHeaderFile', 'memcpy_int.h',...
                                  'SideEffects',            true);
   ```

5. Create conceptual arguments `y1`, `u1`, `u2`, and `u3`. There are multiple ways to set up the conceptual arguments. This example uses calls to the `getTflArgFromString` and `addConceptualArg` functions to create and add the arguments.

   ```
   arg = getTflArgFromString(hTable, 'y1', 'void*');
   arg.IOType = 'RTW_IO_OUTPUT';
   addConceptualArg(fcn_entry, arg);

   arg = getTflArgFromString(hTable, 'u1', 'void*');
   addConceptualArg(fcn_entry, arg);

   arg = getTflArgFromString(hTable, 'u2', 'void*');
   addConceptualArg(fcn_entry, arg);

   arg = getTflArgFromString(hTable, 'u3', 'size_t');
   addConceptualArg(fcn_entry, arg);
   ```

6. Copy the conceptual arguments to the implementation arguments. There are multiple ways to set up the implementation arguments. This example uses a call

to the `copyConceptualArgsToImplementation` function to create and add implementation arguments to the entry by copying matching conceptual arguments.

```
copyConceptualArgsToImplementation(fcn_entry);
```

7 Add the entry to a code replacement table with a call to the `addEntry` function.

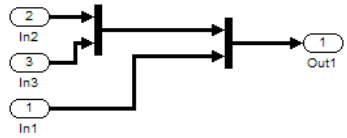
```
addEntry(hTable, fcn_entry);
```

8 Save the table definition file. Use the name of the table definition function to name the file.

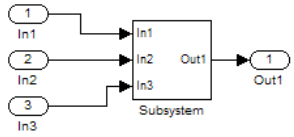
To test this example:

1 Register the code replacement mapping.

2 Create a model that uses the `memcpy` function for vector assignments. For example, use In, Out, and Mux blocks to create the following model. (Alternatively, open the example model rtwdemo_crlmath and copy the contents of Subsystem1 to a new model.)



3 Select the diagram and use **Edit > Subsystem** to make it a subsystem.



4 Configure the subsystem with the following settings:

- On the **Solver** pane, select a fixed-step solver.
- On the **Optimization > Signals and Parameters** pane, select **Use memcpy for vector assignment** and set **Memcpy threshold (bytes)** to 64.
- On the **Code Generation** pane, select an ERT-based system target file.
- On the **Code Generation > Interface** pane, select the code replacement library that contains your memory function entry.

5 In the Model Explorer, configure the **Signal Attributes** for the `In1`, `In2`, and `In3` source blocks. For each, set **Port dimensions** to `[1,100]`, and set **Data type** to `int32`. Apply the changes. Save the model.

6 Generate code and a code generation report.

7 Review the code replacements.

More About

- "Code You Can Replace From Simulink Models" on page 51-7
- "Define Code Replacement Mappings" on page 51-42
- "Data Alignment for Code Replacement" on page 51-133
- "Reserved Identifiers and Code Replacement" on page 51-152
- "Customize Match and Replacement Process" on page 51-153
- "Develop a Code Replacement Library" on page 51-27

Nonfinite Function Code Replacement

This example shows how to define a code replacement mapping for nonfinite utility functions.

1. Create a table definition file that contains a function definition. For example:

   ```
   function hTable = crl_table_nonfinite()
   ```

2. Within the function body, create the table by calling the function `RTW.TflTable`.

   ```
   hTable = RTW.TflTable;
   ```

3. Create entries for the function mappings. To minimize the size of this function, the example uses a local function, `locAddFcnEnt`, to group lines of code repeated for each entry. A call to the `RTW.TflCFunctionEntry` function creates an entry for the collection of local function entry definitions.

   ```
   %% Create entries for nonfinite utility functions
   % locAddFcnEnt(hTable, key, implName, out, in1, hdr)

   locAddFcnEnt(hTable, 'getNaN', 'getNaN', 'double', 'void', 'nonfin.h');
   locAddFcnEnt(hTable, 'getNaN', 'getNaNF', 'single', 'void', 'nonfin.h');
   locAddFcnEnt(hTable, 'getInf', 'getInf',  'double', 'void', 'nonfin.h');
   locAddFcnEnt(hTable, 'getInf', 'getInfF', 'single', 'void', 'nonfin.h');
   locAddFcnEnt(hTable, 'getMinusInf', 'getMinusInf', 'double', 'void', 'nonfin.h');
   locAddFcnEnt(hTable, 'getMinusInf', 'getMinusInfF', 'single', 'void', 'nonfin.h');

   %% Local Function
   function locAddFcnEnt(hTable, key, implName, out, in1, hdr)
     if isempty(hTable)
       return;
     end

    fcn_entry = RTW.TflCFunctionEntry;
   ```

4. Set function entry parameters with a call to the `setTflCFunctionEntryParameters` function.

   ```
    setTflCFunctionEntryParameters(fcn_entry, ...
                                   'Key', key, ...
                                   'Priority', 90, ...
                                   'ImplementationName', implName, ...
                                   'ImplementationHeaderFile', hdr);
   ```

5. Create conceptual arguments `y1` and `u1`. There are multiple ways to set up the conceptual arguments. This example uses calls to the `getTflArgFromString` and `addConceptualArg` functions to create and add the arguments.

   ```
     arg = getTflArgFromString(hTable, 'y1', out);
     arg.IOType = 'RTW_IO_OUTPUT';
     addConceptualArg(fcn_entry, arg);
   ```

```
arg = getTflArgFromString(hTable, 'u1', in1);
addConceptualArg(fcn_entry, arg);
```

6 Copy the conceptual arguments to the implementation arguments. There are multiple ways to set up the implementation arguments. This example uses a call to the `copyConceptualArgsToImplementation` function to create and add implementation arguments to the entry by copying matching conceptual arguments.

```
copyConceptualArgsToImplementation(fcn_entry);
```

7 Add the entry to a code replacement table with a call to the `addEntry` function.

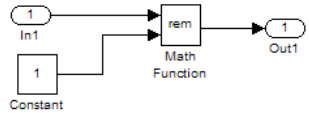
```
addEntry(hTable, fcn_entry);
```

8 Save the table definition file. Use the name of the table definition function to name the file.

To test this example:

1 Register the code replacement mapping.

2 Create a model that uses a nonfinite function. For example, create a model that includes a Math Function block that is set to the `rem` function. For example:



3 Configure the model with the following settings:

- On the **Solver** pane, select a fixed-step solver.
- On the **Code Generation** pane, select an ERT-based system target file.
- On the **Code Generation** > **Interface** pane, select the code replacement library that contains your memory function entry and select **Support: non-finite numbers**.

4 In the Model Explorer, configure the **Signal Attributes** for the `In1` and `Constant` source blocks. For each source block, set **Data type** to `double`. Apply the changes. Save the model.

5 Generate code and a code generation report.

6 Review the code replacements.

More About

- “Code You Can Replace From Simulink Models” on page 51-7

- “Define Code Replacement Mappings” on page 51-42
- “Data Alignment for Code Replacement” on page 51-133
- “Reserved Identifiers and Code Replacement” on page 51-152
- “Customize Match and Replacement Process” on page 51-153
- “Develop a Code Replacement Library” on page 51-27

Semaphore and Mutex Function Replacement

You can create a code replacement table for a custom target that supports concurrent execution. Create table entries that specify custom implementations of semaphore or mutex operations. The table must have four semaphore entries, four mutex entries, or both, and include the table in a custom code replacement library. (The semaphore or mutex entries are mutually dependent. Provide them in complete sets of four.)

Note: A custom target that supports concurrent multitasking must set the target configuration parameter `ConcurrentExecutionCompliant`. For more information, see "Support Concurrent Execution of Multiple Tasks" (Simulink Coder).

If the build process generates semaphore or mutex function calls for data transfer between tasks during code generation for a multicore target environment, use a custom library. The library can specify code replacements for custom semaphore or mutex implementations that are optimal for your target environment. Using the Code Replacement Tool (`crtool`) or equivalent code replacement functions, you can:

- Configure code replacement table entries for custom semaphore or mutex functions. During system startup, execution of the code for data transfer between tasks, and system shutdown the generated code calls these functions.
- Configure DWork arguments that represent global data, which the semaphore or mutex functions access. A DWork pointer is passed to the model entry functions.

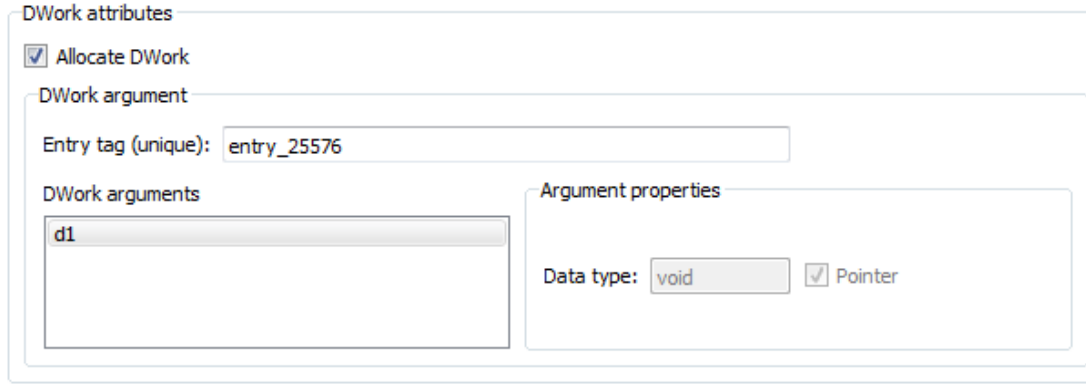
Generated mutex and semaphore code typically consists of these elements:

Code	Generated Code
Model initialization	Initialization function call that creates a mutex or semaphore function to control entry to a critical section of code.
Model step	• Before code for a data transfer between tasks enters the critical section, mutex lock or semaphore wait function calls reserve the critical section of code. • After code for a data transfer between tasks finishes executing the critical section, mutex unlock or semaphore post function calls release the critical section of code.

Code	Generated Code
Model termination	Optional destroy function call to delete the mutex or semaphore explicitly.

This example shows how to create code replacement table entries for a mutex replacement scenario. You configure a multicore target model for concurrent execution and for data transfer between tasks of differing rates, which Rate Transition blocks handle. In the generated code for the model, each Rate Transition block has a separate, unique mutex. Mutex lock and unlock operations within the Rate Transition block generated code share access to the same global data. They achieve this by using the unique mutex created for that Rate Transition block.

1. Open the Code Replacement Tool.
2. Create and open a new table.
3. Name the table `crl_table_rt_mutex`.
4. Create an entry for a mutex initialization function replacement.
   a. Select **File** > **New entry** > **Semaphore entry** to open a new table entry for configuring a semaphore or mutex replacement.
   b. In the **Mapping Information** tab, use the **Function** parameter to select `Mutex Init`. Initial default values for the table entry appear. In the **Conceptual function** section, typically you can leave the argument settings at their defaults.
   c. In the **DWork attributes** section, the **Allocate DWork** option is selected. The dialog box provides a unique entry tag for the DWork argument `d1`.



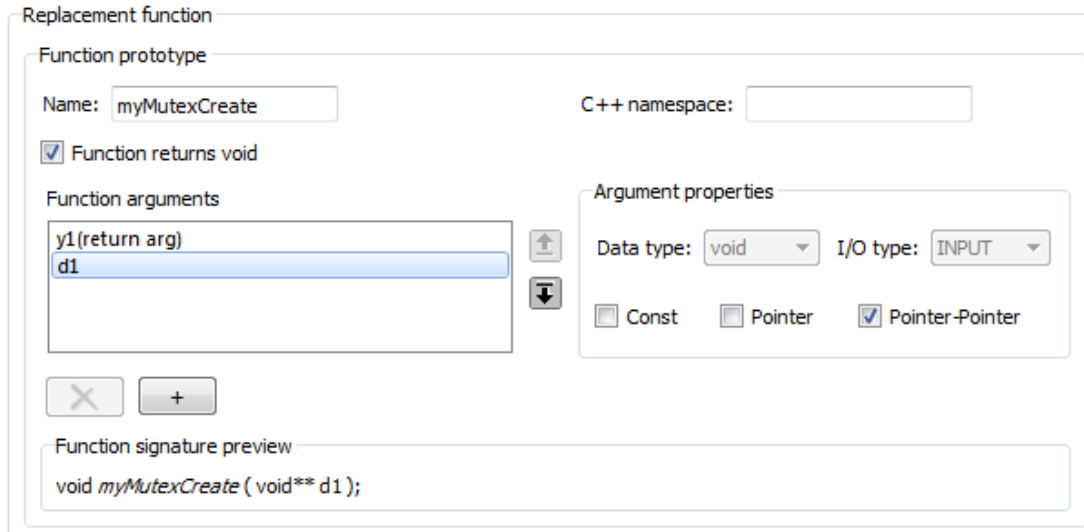
On the **DWork attributes** pane, configure a DWork argument to the replacement function. The DWork argument supports sharing of a semaphore or mutex between:

- Code that creates the semaphore or mutex
- Code that requests and relinquishes access
- Code that deletes the semaphore or mutex

In this example, the DWork argument for the `Mutex Init` function defines a unique entry tag, `entry_25576`. That function also defines DWork arguments for `Mutex Lock`, `Mutex Unlock`, and `Mutex Destroy`, which reference the entry tag to share the DWork data.

The only data type supported for the DWork **Data type** parameter is `void*`.

d In the **Replacement function** section, enter a function name in the **Name** field. This example uses `myMutexCreate`. In the list of **Function arguments**, leave the DWork argument `d1` data type as `void**`.



The C function signature preview is:

```
void myMutexCreate (void** d1);
```

e In the **Replacement function** section, select **Function modifies internal or global state**. This option instructs the code generator not to optimize away the implementation function described by this entry because it accesses global memory values. Click **Apply**. Optionally, you can click **Validate entry** to validate the information entered in the **Mapping Information** tab.

   To create a sample table entry, configure the replacement function signature without the replacement function and its build information. If header and source files for these functions are available, select the **Build Information** table to specify them.

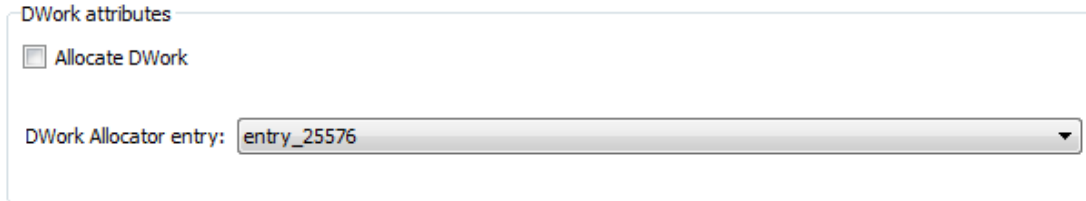
f The `Mutex Init` table entry is complete. Optionally, you can save the table to a file, and inspect the MATLAB code created for the table definition so far.

5 Repeat the following sequence to create the table entries for the mutex lock, unlock, and destroy function replacements. Each table entry references the DWork unique tag entry, `entry_25576`, defined in the `Mutex Init` table entry.

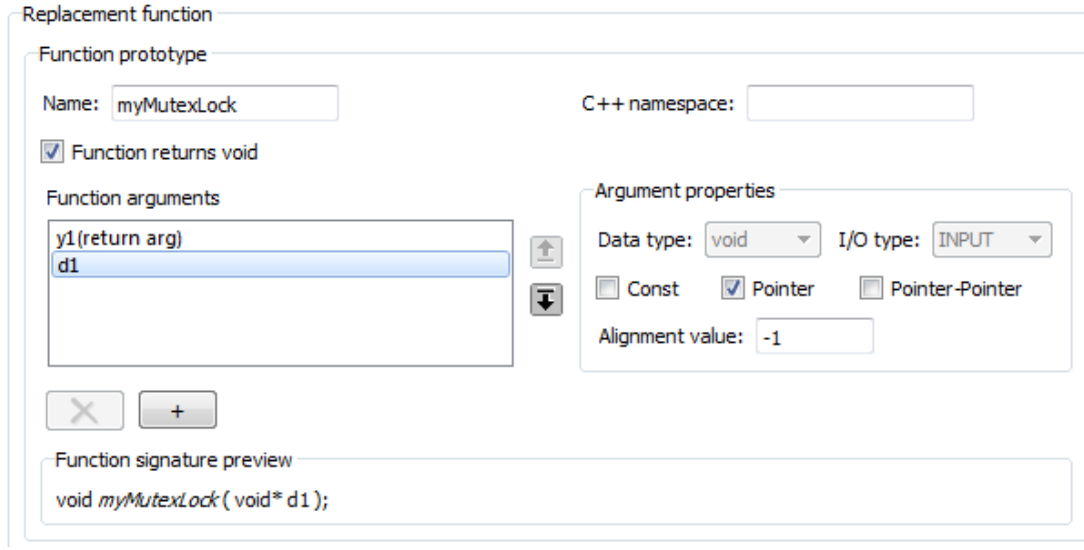
a Select **File** > **New entry** > **Semaphore entry**.

b In the **Mapping Information** tab, use the **Function** parameter to select `Mutex Lock`, `Mutex Unlock`, or `Mutex Destroy`. Initial default values for the table entry appear. In the **Conceptual function** section, typically you can leave the argument settings at their defaults.

c For a Rate Transition block mutex, the wait, post, and destroy functions operate on the DWork allocated at system startup by the mutex initialization function. In the **DWork attributes** section, verify that the **Allocate DWork** option is cleared. From the **DWork Allocator entry** drop-down list, select the entry tag matching the value in the `Mutex Init` table entry. In this example, the entry tag is `entry_25576`.



d In the **Replacement function** section, **Name** field, enter a function name. This example uses `myMutexLock`, `myMutexUnlock`, and `myMutexDelete`. In the list of **Function arguments**, leave the DWork argument `d1` data type as `void*`.

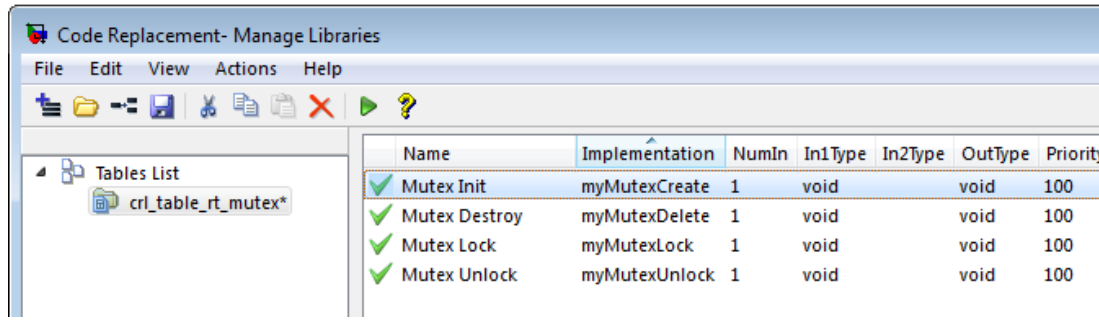


e In the **Implementation attributes** section, select the option **Function modifies internal or global state**. This option instructs the code generator not to optimize away the implementation function described by this entry because it accesses global memory values.

f Optionally, supply build information for the replacement function on the **Build Information** tab.

g Click **Apply**. In the middle pane, right-click the table entry and select **Validate entry(s)**.

6 When you have added the table entries for `Mutex Lock`, `Mutex Unlock`, and `Mutex Destroy` to the entry for `Mutex Init`, the rate transition mutex replacement table is complete. In the left-most pane, right-click the table name and select **Validate table**. Address errors and repeat the table validation.

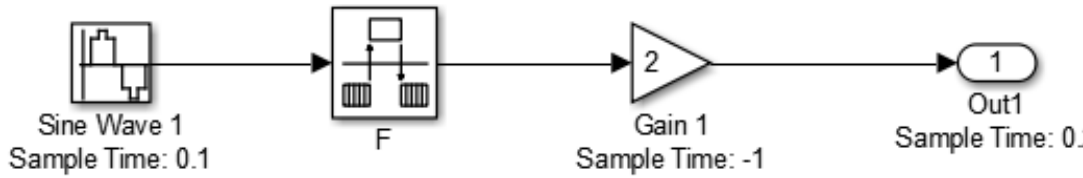


7 Save the table to a MATLAB file in your working folder, for example, using **File** > **Save table**. The name of the saved file is the table name, `crl_table_rt_mutex`, with an `.m` extension. Optionally, you can open the saved file and examine the MATLAB code for the code replacement table definition.

To test this example:

1 Register the code replacement mapping.

2 Create a model that contains a rate transition for which the build process generates mutex function calls. For example:



3 Configure the model for a multicore target environment and the following settings:

- On the **Solver** pane, select a fixed-step solver.
- On the **Code Generation** pane, select an ERT-based system target file.
- On the **Code Generation** > **Interface** pane, select the code replacement library that contains your mutex entry.

More About

- “Code You Can Replace From Simulink Models” on page 51-7
- “Define Code Replacement Mappings” on page 51-42

- "Data Alignment for Code Replacement" on page 51-133
- "Reserved Identifiers and Code Replacement" on page 51-152
- "Customize Match and Replacement Process" on page 51-153
- "Develop a Code Replacement Library" on page 51-27

Algorithm-Based Code Replacement

For some math function blocks, you can control code replacement based on the computation or approximation algorithm configured for that block. For example, you can configure:

- The Reciprocal Sqrt block to use the `Newton-Raphson` or `Exact` computation method.
- The Trigonometric Function block, with **Function** set to `sin`, `cos`, or `sincos`, to use the approximation method `CORDIC` or `None`.

You can define code replacement entries to replace these functions for one or all of the available computation methods. For example, you can define an entry to replace only `Newton-Raphson` instances of the `rSqrt` function.

To set the algorithm for a function in an entry definition, use the `EntryInfoAlgorithm` property in a call to the function `setTflCFunctionEntryParameters`. The following table lists arguments for specifying the computation method to match during code generation.

Function	Argument
`rSqrt`	• `'RTW_DEFAULT'` (match the default computation method, `Exact`) • `'RTW_NEWTON_RAPHSON'` • `'RTW_UNSPECIFIED'` (match any computation method)

`sin`
`cos`
`sincos`
	• `'RTW_CORDIC'`
	• `'RTW_DEFAULT'` (match the default approximation method, `None`)
	• `'RTW_UNSPECIFIED'` (match any approximation method)

For example, to replace only `Newton-Raphson` instances of the `rSqrt` function, create an entry as follows:

1. Create a table definition file that contains a function definition. For example:

```
function hTable = crl_rsqrt()
%CRL_TABLE_RSQRT - Define function entry for code replacement table.
```

2. Within the function body, create the table by calling the function `RTW.TflTable`.

```
hTable = RTW.TflTable;
```

3 Create an entry for the function mapping with a call to the `RTW.TflCFunctionEntry` function.

```
% Create entry for rsqrt function replacement
fcn_entry = RTW.TflCFunctionEntry;
```

4 Set function entry parameters with a call to the `setTflCFunctionEntryParameters` function.

```
setTflCFunctionEntryParameters(fcn_entry, ...
  'Key',                    'rSqrt', ...
  'Priority',               80, ...
  'ImplementationName',     'rsqrt_newton', ...
  'ImplementationHeaderFile', 'rsqrt.h', ...
  'EntryInfoAlgorithm',     'RTW_NEWTON_RAPHSON');
```

5 Create conceptual arguments `y1` and `u1`. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call.

```
createAndAddConceptualArg(fcn_entry, 'RTW.TflArgNumeric', ...
  'Name',         'y1', ...
  'IOType',       'RTW_IO_OUTPUT', ...
  'DataTypeMode', 'double');

createAndAddConceptualArg(e, 'RTW.TflArgNumeric', ...
  'Name',         'u1', ...
  'DataTypeMode', 'double');
```

6 Copy the conceptual arguments to the implementation arguments. This example uses a call to the `copyConceptualArgsToImplementation` function to create and add implementation arguments to the entry by copying matching conceptual arguments.

```
copyConceptualArgsToImplementation(fcn_entry);
```

7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, fcn_entry);
```

8 Save the table definition file. Use the name of the table definition function to name the file.

The generated code for a `Newton-Raphson` instance of the `rSqrt` function looks like the following code:

```
/* Model step function */
```

```
void mrsqrt_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Inport: '<Root>/In1'
   *  Sqrt:   '<Root>/rSqrtBlk'
   */
  mrsqrt_Y.Out1 = rsqrt_newton(mrsqrt_U.In1);
}
```

More About

- “Code You Can Replace From Simulink Models” on page 51-7
- “Math Function Code Replacement” on page 51-94
- “Define Code Replacement Mappings” on page 51-42
- “Develop a Code Replacement Library” on page 51-27

Lookup Table Function Code Replacement

You can configure the algorithm for table lookup operations and index searches to better meet your application code requirements. Use the **Algorithm** tab of lookup table blocks. For example, you can specify the interpolation, extrapolation, and index search methods.

Lookup Table Algorithm Replacement

If the code generated for available algorithm options does not meet requirements for your application, create custom code replacement table entries to replace generated algorithm code. You can create the table entries programmatically or interactively by using the Code Replacement Tool.

For more information about using lookup table blocks, see "Nonlinearity" (Simulink).

Lookup Table Function Signatures

To create code replacement table entries for a function corresponding to a lookup table algorithm, you must have:

- Information about the conceptual function signature.
- Relevant algorithm parameters.

The following table provides the conceptual function signature information.

Conceptual Function Signature	Argument Summary
`y1 = interp1D(u1, u2, u3, u4)`	`y1` – output `u1` – index `u2` – fraction `u3` – table data `u4` – table dimension length
`y1 = interp2D(u1, u2, u3, u4, u5, u6, u7)`	`y1` – output `u1, u3` – index `u2, u4` – fraction `u5` – table data `u6, u7` – table dimension lengths
`y1 = interp3D(u1, u2, u3, u4, u5, u6, u7, u8, u9, u10)`	`y1` – output `u1, u3, u5` – index

Conceptual Function Signature	Argument Summary
	`u2, u4, u6` – fraction `u7` – table data `u8, u9, u10` – table dimension lengths
`y1 = interp4D(u1, u2, u3, u4, u5, u6, u7, u8, u9, u10, u11, u12, u13)`	`y1` – output `u1, u3, u5, u7` – index `u2, u4, u6, u8` – fraction `u9` – table data `u10, u11, u12, u13` – table dimension lengths
`y1 = interp5D(u1, u2, u3, u4, u5, u6, u7, u8, u9, u10, u11, u12, u13, u14, u15, u16)`	`y1` – output `u1, u3, u5, u7, u9` – index `u2, u4, u6, u8, u10` – fraction `u11` – table data `u12, u13, u14, u15, u16` – table dimension lengths
`y1 = interpND({ui, uf,}... ut, un...)`	`y1` – output `ui`, `uf` is an index and fraction pair per dimension `ut` – table data `un` – table dimension lengths
Explicit values `y1 = lookup1D(u1, u2, u3, u4)`	`y1` – output `u1` – input `u2` – breakpoint data `u3` – table data `u4` – table dimension length
Even spacing `y1 = lookup1D(u1, u2, u3, u4, u5)`	`y1` – output `u1` – input `u2` – first point of breakpoint data `u3` – spacing of breakpoints `u4` – table data `u5` – table dimension length

Conceptual Function Signature	Argument Summary
Explicit values `y1 = lookup2D(u1, u2, u3, u4, u5, u6, u7)`	`y1` – output `u1, u2` – input `u3, u4` – breakpoint data `u5` – table data `u6, u7` – table dimension lengths
Even spacing `y1 = lookup2D(u1, u2, u3, u4, u5, u6, u7, u8, u9)`	`y1` – output `u1, u2` – input `u3, u5` – first point of breakpoint data `u4, u6` – spacing of breakpoints `u7` – table data `u8, u9` – table dimension lengths
Explicit spacing `y1 = lookup3D(u1, u2, u3, u4, u5, u6, u7, u8, u9, u10)`	`y1` – output `u1, u2, u3` – input `u4, u5, u6` – breakpoint data `u7` – table data `u8, u9, u10` – table dimension lengths
Even spacing `y1 = lookup3D(u1, u2, u3, u4, u5, u6, u7, u8, u9, u10, u11, u12, u13)`	`y1` – output `u1, u2, u3` – input `u4, u6, u8` – first point of breakpoint data `u5, u7, u9` – spacing of breakpoints `u10` – table data `u11, u12, u13` – table dimension lengths
Explicit values `y1 = lookup4D(u1, u2, u3, u4, u5, u6, u7, u8, u9, u10, u11, u12, u13)`	`y1` – output `u1, u2, u3, u4` – input `u5, u6, u7, u8` – breakpoint data `u9` – table data `u10, u11, u12, u13` – table dimension lengths

Conceptual Function Signature	Argument Summary
Even spacing `y1 = lookup4D(u1, u2, u3, u4, u5, u6, u7, u8, u9, u10, u11, u12, u13, u14, u15, u16, u17)`	`y1` – output `u1, u2, u3, u4` – input `u5, u7, u9, u11` – first point of breakpoint data `u6, u8, u10, u12` – spacing of breakpoints `u13` – table data `u14, u15, u16, u17` – table dimension lengths
Explicit values `y1 = lookup5D(u1, u2, u3, u4, u5, u6, u7, u8, u9, u10, u11, u12, u13, u14, u15, u16)`	`y1` – output `u1, u2, u3, u4, u5` – input `u6, u7, u8, u9, u10` – breakpoint data `u11` – table data `u12, u13, u14, u15, u16` – table dimension lengths
Even spacing `y1 = lookup5D(u1, u2, u3, u4, u5, u6, u7, u8, u9, u10, u11, u12, u13, u14, u15, u16, u17, u18, u19, u20, u21)`	`y1` – output `u1, u2, u3, u4, u5` – input `u6, u8, u10, u12, u14` – first point of breakpoint data `u7, u9, u11, u13, u15` – spacing of breakpoints `u16` – table data `u17, u18, u19, u20, u21` – table dimension lengths
Explicit values `y1 = lookupND(u`*n*`,..., u`*b*`,..., u`*t*`, u`*n*`...)`	`y1` – output `u`*n*, input per dimension `u`*b*, breakpoint per dimension `u`*t* – table data `u`*n* – table dimension lengths

Conceptual Function Signature	Argument Summary
Even spacing `y1 = lookupND(un,..., {ufn, usn,}... ut, un...)`	`y1` – output `un` – input per dimension `ufn` – first point of breakpoint data per dimension `usn` – spacing of breakpoint per dimension `ut` – table data `un` – table dimension lengths
`y1 = lookupND_Direct(u1, u2,...ui, ui+1)`	`y1` – output `u1...ui` – input `ui+1` – table data
Explicit values `y1, y2 = prelookup(u1, u2, u3)`	`y1` – index `y2` – fraction `u1` – input `u2` – breakpoint data `u3` – number of breakpoints
Evenly spaced `y1, y2 = prelookup(u1, u2, u3, u4)`	`y1` – index `y2` – fraction `u1` – input `u2` – first point of breakpoint data `u3` – spacing of breakpoints `u4` – number of breakpoints

When defining a table entry programmatically, you might also need to change the values of required (primary) and optional algorithm parameters.

- Set values for required parameters to achieve code replacement.
- If you do not set a value for an optional parameter, the algorithm parameter software applies `don't care`. The code replacement software ignores the parameter while searching for matches.

To look up algorithm parameter information for a lookup table function:

1. Create a table entry for a function.

   ```
   tableEntry = RTW.TflCFunctionEntry;
   ```

2 Identify the lookup table function in the table entry. Use the `Key` table entry parameter in a call to `setTflCFunctionEntryParameters`. The following example identifies an entry for the `prelookup` function.

```
setTflCFunctionEntryParameters(tableEntry, ...
    'Key', 'prelookup', ...
    'Priority', 100, ...
    'ImplementationName', 'myPrelookup');
```

3 Get the algorithm parameter set for the entry with a call to `getAlgorithmParameters`.

```
algParams = getAlgorithmParameters(tableEntry);

algParams =

  Prelookup with properties:

            ExtrapMethod: [1x1 coder.algorithm.parameter.ExtrapMethod]
                 RndMeth: [1x1 coder.algorithm.parameter.RndMeth]
       IndexSearchMethod: [1x1 coder.algorithm.parameter.IndexSearchMethod]
       UseLastBreakpoint: [1x1 coder.algorithm.parameter.UseLastBreakpoint]
   RemoveProtectionInput: [1x1 coder.algorithm.parameter.RemoveProtectionInput]
```

4 Examine information available for each parameter.

```
algParams.ExtrapMethod

ans =

  ExtrapMethod with properties:

       Name: 'ExtrapMethod'
    Options: {'Linear'  'Clip'}
    Primary: 1
      Value: {'Linear'}

algParams.RndMeth

ans =

  RndMeth with properties:

       Name: 'RndMeth'
    Options: {1x7 cell}
    Primary: 0
      Value: {1x7 cell}

algParams.RndMeth.Value

ans =

  Columns 1 through 6

    'Ceiling'    'Convergent'    'Floor'    'Nearest'    'Round'    'Simplest'

  Column 7

    'Zero'

algParams.IndexSearchMethod
```

```
ans =

  IndexSearchMethod with properties:

       Name: 'IndexSearchMethod'
    Options: {'Linear search'  'Binary search'  'Evenly spaced points'}
    Primary: 0
      Value: {'Binary search'  'Evenly spaced points'  'Linear search'}

algParams.UseLastBreakpoint

ans =

  UseLastBreakpoint with properties:

       Name: 'UseLastBreakpoint'
    Options: {'off'  'on'}
    Primary: 0
      Value: {'off'  'on'}

algParams.RemoveProtectionInput

ans =

  RemoveProtectionInput with properties:

       Name: 'RemoveProtectionInput'
    Options: {'off'  'on'}
    Primary: 0
      Value: {'off'  'on'}
```

Interactive Mapping with Code Replacement Tool

This example shows how to specify a code replacement table entry for a lookup table algorithm by using the Code Replacement Tool.

Open and Examine Example Replacement Function

Identify or create the C or C++ replacement function for the algorithm that you want to use in place of a Simulink software algorithm.

This example uses the following C replacement function header and source files, which are in the folder `matlab/toolbox/rtw/rtwdemos/crl_demo`:

- `myLookup1D.h`
- `myLookup1D.c`

Place a copy of these files in your working folder.

Open and examine the code for `myLookup1D.h`.

```
#include "rtwtypes.h"
real_T my_Lookup1D_Repl(real_T u0, const real_T *bp0, const real_T *table, uint32_T tdl);
```

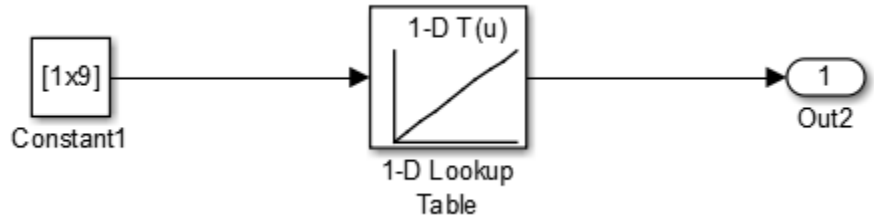
Open and examine the code in `myLookup1D.c`. Note the function signature. When you enter the implementation argument specification in the Code Replacement Tool, specify argument properties.

```
#include "myLookup1D.h"
real_T my_Lookup1D_Repl(real_T u0, const real_T *bp0, const real_T *table, uint32_T tdl)
{
  real_T y;
  uint16_T frac;
  uint32_T bpIdx;
  uint32_T maxIndex=tdl-1;

  if (u0 <= bp0[0U]) {
    bpIdx = 0U;
    frac = 0U;
  } else if (u0 < bp0[maxIndex]) {
    bpIdx = maxIndex >> 1U;
    while ((u0 < bp0[bpIdx]) && (bpIdx > 0U)) {
      bpIdx--;
    }

    while ((bpIdx < maxIndex - 1U) && (u0 >= bp0[bpIdx + 1U])) {
      bpIdx++;
    }

    frac = (uint16_T)((u0 - bp0[bpIdx]) / (bp0[bpIdx + 1U] -
      bp0[bpIdx]) * 32768.0);
  } else {
    bpIdx = maxIndex;
    frac = 0U;
  }

  if (bpIdx == maxIndex) {
    y = table[bpIdx];
  } else {
    y = (table[bpIdx + 1U] - table[bpIdx]) * ((real_T)frac * 3.0517578125E-5) +
      table[bpIdx];
  }

  return y;
}
```

Open and Examine the Example Model

This example uses the model rtwdemo_crllookup1D to test your code replacement specification. Place a copy of the model in your working folder and name it `my_lookup1d.slx`.

Open and examine the model. Note input and output specifications and block parameter settings. To achieve a match, you must specify conceptual arguments based on how the 1-D Lookup Table block is configured in the example model.



Create Code Replacement Table

1. At the command prompt, enter `crtool` to open the Code Replacement Tool.
2. Add a new table, select that table, and add a new function entry.
3. On the **Mapping Information** tab, select `Custom` for the **Function** parameter.
4. Look up the call signature and algorithm parameter information for the lookup table function that you want to update with an algorithm replacement. See "Lookup Table Function Signatures" on page 51-112.

   For this example, you replace the algorithm for the conceptual function associated with the 1-D Lookup Table block. The signature for that function is:

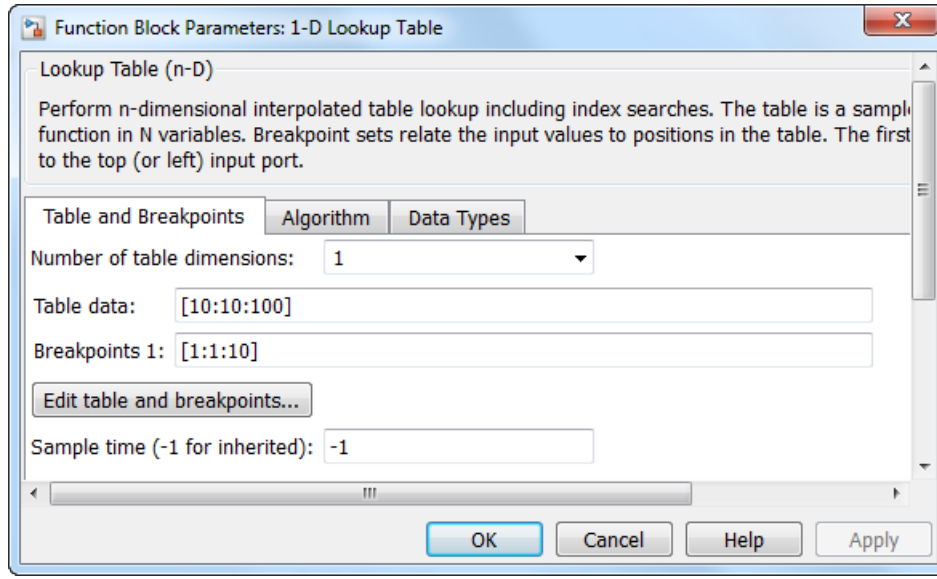
   ```
   y1 = lookup1D(u1, u2, u3, u4)
   ```

   Arguments `u1`, `u2`, `u3`, `u4` represent input, breakpoint data, table data, and table dimension length, respectively. The function returns output to `y1`.
5. To the right of the **Function** drop-down list, in the function-name text box, enter the name of the Simulink lookup table function. For this example, type the name `lookup1D`. Type the name exactly as it appears in the documented signature, including character casing. Press **Enter**.

   The tool displays algorithm parameter settings that trigger a match for the 1-D Lookup Table block in the example model. Required parameters appear with only one value. For this example, do not change the values. Optional parameters appear with multiple values. Changes to optional parameters do not affect the match process.
6. Specify the conceptual arguments. Under the **Conceptual arguments** list box, click **+** to add the arguments that are in the documented function signature. The `lookup1D` function takes one output argument and four input arguments. Click **+** five times.

The tool creates an output argument *y1* and four input arguments *u1*, *u2*, *u3*, and *u4*. By default, the four arguments are scalars of type `double`.

You can adjust the conceptual argument properties. For this example, you do not make changes for `y1` and `u1`. However, as the block parameter dialog box for the example model shows, you must adjust the argument properties for the breakpoint and table data arguments.



Adjust the conceptual argument properties by using the following table. Click **Apply**.

Signature Argument Name	Conceptual Argument Name	Data type	I/O type	Argument type	Lower range	Upper range
`y`	`y1`	`double`	`OUTPUT`	`Scalar`	Not applicable	Not applicable
`u1`	`u1`	`double`	`INPUT`	`Scalar`	Not applicable	Not applicable
`bp1`	`u2`	`double`	`INPUT`	`Matrix`	`[0 0]`	`[Inf Inf]`

Signature Argument Name	Conceptual Argument Name	Data type	I/O type	Argument type	Lower range	Upper range
`table`	`u3`	`double`	`INPUT`	`Matrix`	`[0 0]`	`[Inf Inf]`
`tdl`	`u4`	`uint32`	`INPUT`	`Scalar`	Not applicable	Not applicable

7 Enter information for the replacement function prototype. The prototype for the example function is:

```
real_T my_Lookup1D_Repl(real_T u0, const real_T *bp0, const real_T *table, uint32_T tdl)
```

In the **Replacement function** > **Function prototype** section, type the function name `my_Lookup1D_Repl` in the **Name** text box.

8 Specify the arguments for the replacement function. Under the **Function arguments** list box, click **+** five times to add five implementation arguments.

You might need to adjust the function argument properties. As the replacement function signature shows, adjust the argument properties for the breakpoint, table data, and table dimension length arguments. For `u2` (breakpoints) and `u3` (table), select the **Const** check box. For `u4`, set **Data type** to `uint32`.

The function signature preview should appear as follows:

```
double my_Lookup1D_Repl(double u1, const double* u2, const double* u3, uint32 u4)
```

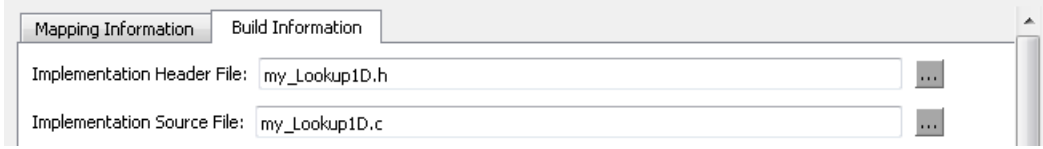
9 Set relevant implementation attributes. Use the default settings.

10 Validate the entry. If the tool reports errors, fix them, and retry the validation. Repeat the procedure until the tool does not report errors.

11 Save the code replacement table in your working folder as `my_lookup_replacement_table.m`.

Specify Build Information

On the **Build Information** tab, specify information relevant to generating C or C++ code and building an executable from the model. Enter `myLookup1D.h` for **Implementation Header File** and `myLookup1D.c` for **Implementation Source File**.



If you copied the example files to a folder other than the working folder containing the test model, `lookup1d.slx`, specify the source and header file paths. Otherwise, leave the other **Build Information** parameters set to default values. Click **Apply**.

Test the Entry

To test this example:

1. Register the code replacement mapping.
2. Use the example model rtwdemo_crllookup1D.
3. Configure the model with the following settings:
   - On the **Solver** pane, select a fixed-step solver.
   - On the **Code Generation** pane, select an ERT-based system target file.
   - On the **Code Generation** > **Interface** pane, select the code replacement library that contains your lookup table function entry.

Programmatic Specification

This example shows how to specify code replacement table entries for lookup table functions programmatically.

Open and Examine Example Replacement Function

Identify or create the C or C++ replacement function for the algorithm that you want to use in place of a Simulink software algorithm.

This example uses the following C replacement function header and source files, which are in the folder `matlab/toolbox/rtw/rtwdemos/crl_demo`:

- `myLookup1D.h`
- `myLookup1D.c`

Place a copy of these files in your working folder.

Open and examine the code for `myLookup1D.h`.

```
#include "rtwtypes.h"
real_T my_Lookup1D_Repl(real_T u0, const real_T *bp0, const real_T *table, uint32_T tdl);
```

Open and examine the code in `myLookup1D.c`. Note the function signature. When you enter the implementation argument specification in the Code Replacement Tool, specify argument properties.

```
#include "myLookup1D.h"
real_T my_Lookup1D_Repl(real_T u0, const real_T *bp0, const real_T *table, uint32_T tdl)
{
  real_T y;
  uint16_T frac;
  uint32_T bpIdx;
  uint32_T maxIndex=tdl-1;

  if (u0 <= bp0[0U]) {
    bpIdx = 0U;
    frac = 0U;
  } else if (u0 < bp0[maxIndex]) {
    bpIdx = maxIndex >> 1U;
    while ((u0 < bp0[bpIdx]) && (bpIdx > 0U)) {
      bpIdx--;
    }

    while ((bpIdx < maxIndex - 1U) && (u0 >= bp0[bpIdx + 1U])) {
      bpIdx++;
    }

    frac = (uint16_T)((u0 - bp0[bpIdx]) / (bp0[bpIdx + 1U] -
      bp0[bpIdx]) * 32768.0);
  } else {
    bpIdx = maxIndex;
    frac = 0U;
  }

  if (bpIdx == maxIndex) {
    y = table[bpIdx];
  } else {
    y = (table[bpIdx + 1U] - table[bpIdx]) * ((real_T)frac * 3.0517578125E-5) +
      table[bpIdx];
  }

  return y;
}
```

Review Lookup Function Signature

Look up the call signature information for the lookup function that you want to update with an algorithm replacement. See “Lookup Table Function Signatures” on page 51-112.

Replace the algorithm for the function associated with the 1–D Lookup Table block. The signature for that function is:

```
y1 = lookup1D(u1, u2, u3, u4)
```

Arguments `u1`, `u2`, `u3`, and `u4` represent input, breakpoint data, table data, and table dimension length, respectively. The function returns output to `y1`.

Create Code Replacement Entry

Create a code replacement table file as a MATLAB function, that describes the lookup table function code replacement table entries. Place a copy of the file `matlab/toolbox/rtw/rtwdemos/crl_demo/crl_table_lookup1D.m` in your working folder. This file defines a code replacement table for the C function `my_Lookup1D_Repl`.

Open `crl_table_lookup1D.m` and examine the definition.

1. Create a table definition file that contains a function definition. For example:

   ```
   function hLib = my_lookup_replacement_table
   ```

2. Within the function body, create the table by calling the function `RTW.TflTable`.

   ```
   hLib = RTW.TflTable;
   ```

3. Create an entry for the function mapping with a call to the `RTW.TflCFunctionEntry` function.

   ```
   hEnt = RTW.TflCFunctionEntry;
   ```

4. Set function entry parameters with a call to the `setTflCFunctionEntryParameters` function. The function key, implementation name, and header and source files in the function call identify the Simulink lookup table function name, `lookup1D`, and the following information for replacement function `my_Lookup1D_Repl`:

   - Function name
   - Header file
   - Source file

   Specify the Simulink lookup table function name exactly as it appears in the documented signature, including character casing (see “Lookup Table Function Signatures” on page 51-112). If you copied the example files to a folder other than the working folder that contains the test model, rtwdemo_crllookup1D, specify the source and header file paths.

   ```
   setTflCFunctionEntryParameters(hEnt, ...
             'Key',                    'lookup1D', ...
             'Priority',               100, ...
             'ImplementationName',     'my_Lookup1D_Repl', ...
             'ImplementationHeaderFile', 'myLookup1D.h', ...
             'ImplementationSourceFile', 'myLookup1D.c', ...
             'GenCallback',            'RTW.copyFileToBuildDir');
   ```

5 Create conceptual arguments and add them to the entry. This example uses calls to the `getTflArgFromString` and `addConceptualArg` functions to create and add the arguments.

The example defines five conceptual arguments for the `lookup1D` function, one output argument *`y1`* and four input arguments *`u1`*, *`u2`*, *`u3`*, and *`u4`*. Arguments *`y1`* and *`u1`* are defined as scalar `double` data. Arguments *`u2`* and *`u3`* represent *`bp1`* and *`table`* in the signature and are defined as 1x10 matrices of `double` data. Argument *`u4`* represents *`tdl`* and is defined as scalar of `uint32` data. This definition triggers a match with the example model.

```
arg = hEnt.getTflArgFromString('y1','double');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEnt, arg);

arg = hEnt.getTflArgFromString('u1','double');
addConceptualArg(hEnt, arg);

arg = RTW.TflArgMatrix('u2', 'RTW_IO_INPUT',  'double');
arg.DimRange = [0 0; Inf Inf];
addConceptualArg(hEnt, arg);

arg = RTW.TflArgMatrix('u3', 'RTW_IO_INPUT',  'double');
arg.DimRange = [0 0; Inf Inf];
addConceptualArg(hEnt, arg);

arg = hEnt.getTflArgFromString('u4','uint32');
addConceptualArg(hEnt, arg);
```

6 Review the algorithm parameter information for the lookup function that you want to update with an algorithm replacement. Use the `getAlgorithmParameters` function to display the parameter information.

```
algParams = getAlgorithmParameters(hEnt)

algParams =

  Lookup with properties:

                InterpMethod: [1x1 coder.algorithm.parameter.InterpMethod]
                ExtrapMethod: [1x1 coder.algorithm.parameter.ExtrapMethod]
                     RndMeth: [1x1 coder.algorithm.parameter.RndMeth]
           IndexSearchMethod: [1x1 coder.algorithm.parameter.IndexSearchMethod]
           UseLastTableValue: [1x1 coder.algorithm.parameter.UseLastTableValue]
       RemoveProtectionInput: [1x1 coder.algorithm.parameter.RemoveProtectionInput]
  SaturateOnIntegerOverflow: [1x1 coder.algorithm.parameter.SaturateOnIntegerOverflow]
    SupportTunableTableSize: [1x1 coder.algorithm.parameter.SupportTunableTableSize]
            BPPower2Spacing: [1x1 coder.algorithm.parameter.BPPower2Spacing]
```

Examine the information for each parameter. The `Options` property lists possible values. `Primary` indicates whether a parameter is required (1) or optional (0). The

`Value` property specifies the current value. For required parameters, initially, `Value` is set to the default value for a given lookup table function.

```
algParams.InterpMethod

ans =

  InterpMethod with properties:

       Name: 'InterpMethod'
    Options: {'Linear'  'Flat'  'Nearest'}
    Primary: 1
      Value: {'Linear'}

algParams.RndMeth

ans =

  RndMeth with properties:

       Name: 'RndMeth'
    Options: {1x7 cell}
    Primary: 0
      Value: {1x7 cell}

algParams.RndMeth.Options

ans =

  Columns 1 through 5

    'Ceiling'    'Convergent'    'Floor'    'Nearest'    'Round'

  Columns 6 through 7

    'Simplest'    'Zero'

algParams.RndMeth

ans =

  RndMeth with properties:

       Name: 'RndMeth'
    Options: {1x7 cell}
```

```
    Primary: 0
      Value: {1x7 cell}
.
.
.
```

7 Set the algorithm properties for the `lookup1D` table entry. Assign a value to each parameter. Update the parameter settings for the entry by calling the function `setAlgorithmParameters`. The following parameter settings trigger a match with the example model.

```
algParams.InterpMethod = 'Linear';
algParams.ExtrapMethod = 'Clip';
algParams.RndMeth = 'Round';
algParams.IndexSearchMethod = 'Linear search';
algParams.UseLastTableValue = 'Evenly spaced point';
algParams.RemoveProtectionInput = 'off';
algParams.SaturateOnIntegerOverflow = 'off';
algParams.SupportTunableTableSize = 'off';
algParams.BPPower2Spacing = 'off';
setAlgorithmParameters(hEnt, algParams);

ans =

  RndMeth with properties:

       Name: 'RndMeth'
    Options: {1x7 cell}
    Primary: 0
      Value: {1x7 cell}
.
.
.
```

To verify your changes, call `getAlgorithmParameters` to get the parameter set for the table entry. Examine the value of each parameter.

```
getAlgorithmParameters(hEnt, algParams);
algParams.InterpMethod.Value

ans =

    'Linear'

algParams.ExtrapMethod.Value
```

```
ans =

    'Clip'

algParams.RndMeth.Value

ans =

    'Round'
.
.
.
```

8 Create the implementation arguments and add them to the entry. This example uses calls to the `getTflArgFromString` function to create five implementation arguments that map to arguments in the replacement function prototype: one output argument *y1* and four input arguments *u1*, *u2*, *u3*, and *u4*. The convenience methods `setReturn` and `addArgument` specify whether an argument is a return value or argument. The `addArgument` function also adds each argument to the entry's array of implementation arguments.

```
arg = hEnt.getTflArgFromString('y1','double');
arg.IOType = 'RTW_IO_OUTPUT';
hEnt.Implementation.setReturn(arg);

arg = hEnt.getTflArgFromString('u1','double');
hEnt.Implementation.addArgument(arg);

arg = hEnt.getTflArgFromString('u2','double*');
hEnt.Implementation.addArgument(arg);

arg = hEnt.getTflArgFromString('u3','double*');
hEnt.Implementation.addArgument(arg);

arg = hEnt.getTflArgFromString('u4','uint32');
hEnt.Implementation.addArgument(arg);
```

9 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hLib, hEnt);
```

10 Save the table definition file. Use the name of the table definition function to name the file.

Test the Entry

To test this example:

1. Register the code replacement mapping.
2. Use the example model rtwdemo_crllookup1D.
3. Configure the model with the following settings:
   - On the **Solver** pane, select a fixed-step solver.
   - On the **Code Generation** pane, select an ERT-based system target file.
   - On the **Code Generation** > **Interface** pane, select the code replacement library that contains your lookup table function entry.

Sample Code Replacement Definition for the `lookup2D` Function

The following code shows a replacement definition for the `lookup2D` function.

```
function hLib = my_2dlookup_replacement_table

hLib = RTW.TflTable;

hEnt = RTW.TflCFunctionEntry;
setTflCFunctionEntryParameters(hEnt, ...
          'Key',                      'lookup2D', ...
          'Priority',                 100, ...

          'ImplementationName',       'custom_lookup2d', ...
          'ImplementationHeaderFile', 'custom_lookup2d.h', ...
          'ImplementationSourceFile', 'custom_lookup2d.c', ...
          'GenCallback',              'RTW.copyFileToBuildDir');

% Conceptual Args

arg = hEnt.getTflArgFromString('y1','double');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEnt, arg);

arg = hEnt.getTflArgFromString('u1','double');
addConceptualArg(hEnt, arg);

arg = hEnt.getTflArgFromString('u2','double');
addConceptualArg(hEnt, arg);
```

```
arg = RTW.TflArgMatrix('u3', 'RTW_IO_INPUT',  'double');
arg.DimRange = [1 1; 10 1];
addConceptualArg(hEnt, arg);

arg = RTW.TflArgMatrix('u4', 'RTW_IO_INPUT',  'double');
arg.DimRange = [1 1; 10 1];
addConceptualArg(hEnt, arg);

arg = RTW.TflArgMatrix('u5', 'RTW_IO_INPUT',  'double');
arg.DimRange = [1 1; 10 1];
addConceptualArg(hEnt, arg);

arg = hEnt.getTflArgFromString('u6','uint32');
addConceptualArg(hEnt, arg);

arg = hEnt.getTflArgFromString('u7','uint32');
addConceptualArg(hEnt, arg);

% Algorithm Parameters

addAlgorithmProperty(hEnt, 'ExtrapMethod','Clip');
addAlgorithmProperty(hEnt, 'IndexSearchMethod','Linear search');
addAlgorithmProperty(hEnt, 'InterpMethod','Linear');
addAlgorithmProperty(hEnt, 'RemoveProtectionInput','off');
addAlgorithmProperty(hEnt, 'UseLastTableValue','on');

% Implementation Args

arg = hEnt.getTflArgFromString('y1','double');
arg.IOType = 'RTW_IO_OUTPUT';
hEnt.Implementation.setReturn(arg);

arg = hEnt.getTflArgFromString('u1','double');
hEnt.Implementation.addArgument(arg);

arg = hEnt.getTflArgFromString('u2','double');
hEnt.Implementation.addArgument(arg);

arg = hEnt.getTflArgFromString('u3','double*');
arg.Type.BaseType.ReadOnly = true;
hEnt.Implementation.addArgument(arg);

arg = hEnt.getTflArgFromString('u4','double*');
```

```
arg.Type.BaseType.ReadOnly = true;
hEnt.Implementation.addArgument(arg);

arg = hEnt.getTflArgFromString('u5','double*');
arg.Type.BaseType.ReadOnly = true;
hEnt.Implementation.addArgument(arg);

arg = hEnt.getTflArgFromString('u6','uint32');
hEnt.Implementation.addArgument(arg);

arg = hEnt.getTflArgFromString('u7','uint32');
hEnt.Implementation.addArgument(arg);

hLib.addEntry(hEnt);
```

More About

- "Code You Can Replace From Simulink Models" on page 51-7
- "Define Code Replacement Mappings" on page 51-42
- "Data Alignment for Code Replacement" on page 51-133
- "Reserved Identifiers and Code Replacement" on page 51-152
- "Customize Match and Replacement Process" on page 51-153
- "Develop a Code Replacement Library" on page 51-27

Data Alignment for Code Replacement

Code replacement libraries can align data objects passed into a replacement function to a specified boundary.

Code Replacement Data Alignment

You can take advantage of function implementations that require aligned data to optimize application performance. To configure data alignment for a function implementation:

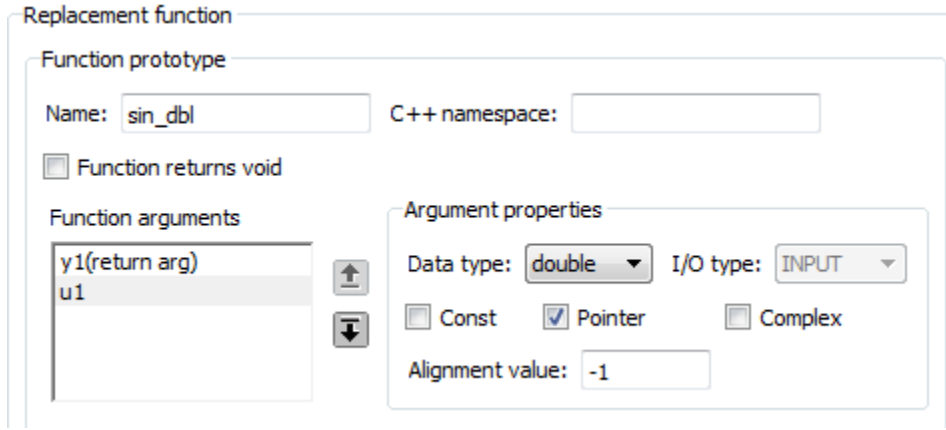
1. Specify the data alignment requirements in a code replacement entry. Specify alignment separately for each implementation function argument or collectively for all function arguments. See "Specify Data Alignment Requirements for Function Arguments" on page 51-133.
2. Specify the data alignment capabilities and syntax for one or more compilers. Include the alignment specifications in a library registration entry in the `rtwTargetInfo.m` file. See "Provide Data Alignment Specifications for Compilers" on page 51-135.
3. Register the library containing the table entry and alignment specification object.
4. Configure the code generator to use the code replacement library and generate code. Observe the results.

For examples, see "Basic Example of Code Replacement Data Alignment" on page 51-139 and the "Data Alignment for Function Implementations" section of the "Optimize Generated Code By Developing and Using Code Replacement Libraries - Simulink®" example page.

Specify Data Alignment Requirements for Function Arguments

To specify the data alignment requirement for an argument in a code replacement entry:

- If you are defining a replacement function in a code replacement table registration file, create an argument descriptor object (`RTW.ArgumentDescriptor`). Use its `AlignmentBoundary` property to specify the required alignment boundary and assign the object to the argument `Descriptor` property.
- If you are defining a replacement function using the Code Replacement Tool, on the **Mapping Information** tab, in the **Argument properties** section for the replacement function, enter a value for the **Alignment value** parameter.



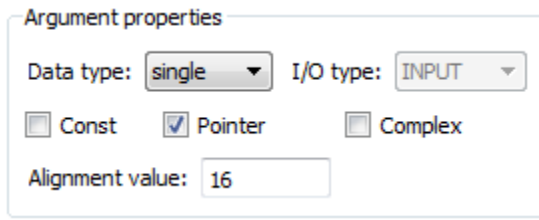
The `AlignmentBoundary` property (or **Alignment value** parameter) specifies the alignment boundary for data passed to a function argument, in number of bytes. The `AlignmentBoundary` property is valid only for addressable objects, including matrix and pointer arguments. It is not applicable for value arguments. Valid values are:

- `-1` (default) — If the data is a `Simulink.Bus`, `Simulink.Signal`, or `Simulink.Parameter` object, specifies that the code generator determines an optimal alignment based on usage. Otherwise, specifies that there is not an alignment requirement for this argument.
- Positive integer that is a power of 2, not exceeding 128 — Specifies number of bytes in the boundary. The starting memory address for the data allocated for the function argument is a multiple of the specified value. If you specify an alignment boundary that is less than the natural alignment of the argument data type, the alignment directive is emitted in the generated code. However, the target compiler ignores the directive.

The following code specifies the `AlignmentBoundary` for an argument as 16 bytes.

```
hLib = RTW.TflTable;
entry = RTW.TflCOperationEntry;
arg = getTflArgFromString(hLib, 'u1','single*');
desc = RTW.ArgumentDescriptor;
desc.AlignmentBoundary = 16;
arg.Descriptor = desc;
entry.Implementation.addArgument(arg);
```

The equivalent alignment boundary specification in the Code Replacement Tool dialog box is in this figure.



Note: If your model imports `Simulink.Bus`, `Simulink.Parameter`, or `Simulink.Signal` objects, specify an alignment boundary in the object properties, using the **Alignment** property. For more information, see `Simulink.Bus`, `Simulink.Parameter`, and `Simulink.Signal`.

Provide Data Alignment Specifications for Compilers

To support data alignment in generated code, describe the data alignment capabilities and syntax for your compilers in the code replacement library registration. Provide one or more alignment specifications for each compiler in a library registry entry.

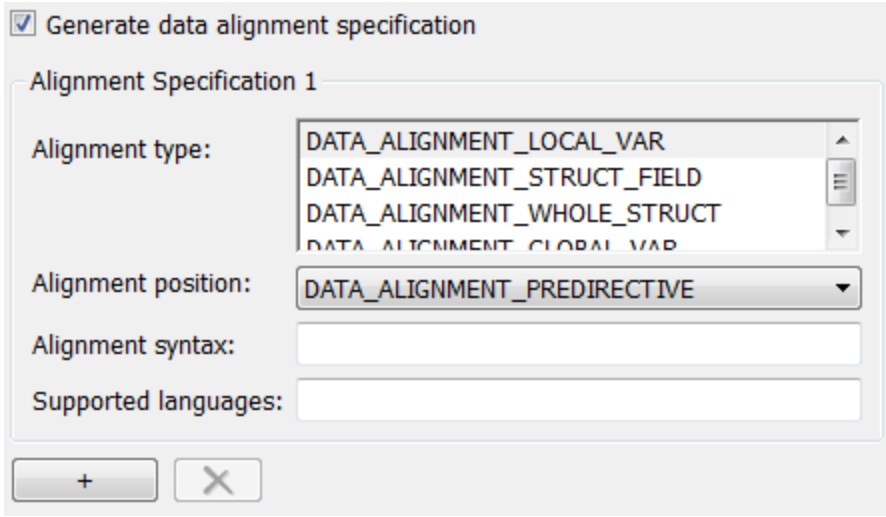
To describe the data alignment capabilities and syntax for a compiler:

- If you are defining a code replacement library registration entry in a `rtwTargetInfo.m` customization file, add one or more `AlignmentSpecification` objects to an `RTW.DataAlignment` object. Attach the `RTW.DataAlignment` object to the `TargetCharacteristics` object of the registry entry.

  The `RTW.DataAlignment` object also has the property `DefaultMallocAlignment`, which specifies the default alignment boundary, in bytes, that the compiler uses for dynamically allocated memory. If the code generator uses dynamic memory allocation for a data object involved in a code replacement, this value determines if the memory satisfies the alignment requirement of the replacement. If not, the code generator does not use the replacement. The default value for `DefaultMallocAlignment` is `-1`, indicating that the default alignment boundary used for dynamically allocated memory is unknown. In this case, the code generator uses the natural alignment of the data type to determine whether to allow a replacement.

Additionally, you can specify the alignment boundary for complex types by using the `addComplexTypeAlignment` function.

- If you are generating a customization file function using the Code Replacement Tool, fill out the following fields for each compiler.



Click the plus (+) symbol to add additional compiler specifications.

For each data alignment specification, provide the following information.

`Alignment-Specification` Property	**Dialog Box Parameter**	**Description**
`AlignmentType`	**Alignment type**	Cell array of predefined enumerated strings, specifying which types of alignment this specification supports. • `DATA_ALIGNMENT_LOCAL_VAR` — Local variables. • `DATA_ALIGNMENT_GLOBAL_VAR` — Global variables. • `DATA_ALIGNMENT_STRUCT_FIELD` — Individual structure fields. • `DATA_ALIGNMENT_WHOLE_STRUCT` — Whole structure, with padding (individual structure field alignment, if

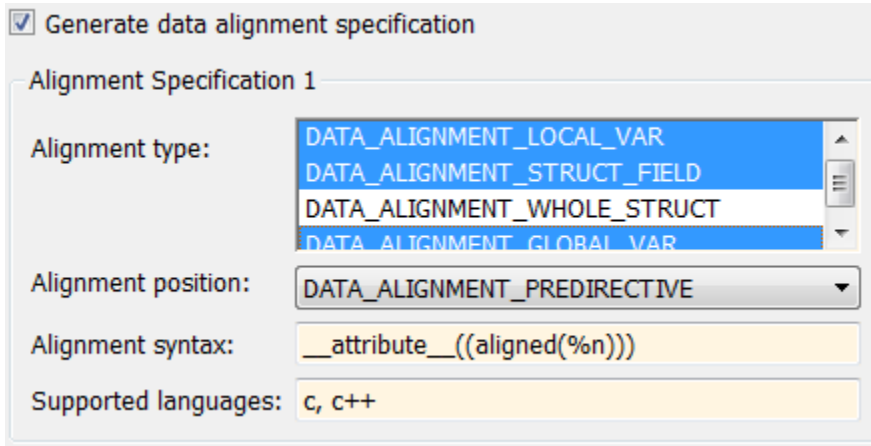
Alignment-Specification Property	Dialog Box Parameter	Description
		specified, is favored and takes precedence over whole structure alignment). Each alignment specification must specify at least DATA_ALIGNMENT_GLOBAL_VAR and DATA_ALIGNMENT_STRUCT_FIELD.
AlignmentPosition	**Alignment position**	Predefined enumerated string specifying the position in which you must place the compiler alignment directive for alignment type DATA_ALIGNMENT_WHOLE_STRUCT: • DATA_ALIGNMENT_PREDIRECTIVE — The alignment directive is emitted before `struct st_tag{…}`, as part of the type definition statement (for example, MSVC). • DATA_ALIGNMENT_POSTDIRECTIVE — The alignment directive is emitted after `struct st_tag{…}`, as part of the type definition statement (for example, gcc). • DATA_ALIGNMENT_PRECEDING_STATEMENT — The alignment directive is emitted as a standalone statement immediately preceding the definition of the structure type. A semicolon (;) must terminate the registered alignment syntax. • DATA_ALIGNMENT_FOLLOWING_STATEMENT — The alignment directive is emitted as a standalone statement immediately following the definition of the structure type. A semicolon (;) must terminate the registered alignment syntax. For alignment types other than DATA_ALIGNMENT_WHOLE_STRUCT, code generation uses alignment position DATA_ALIGNMENT_PREDIRECTIVE.

Alignment-Specification Property	Dialog Box Parameter	Description
`AlignmentSyntax-Template`	**Alignment syntax**	Specifies the alignment directive string that the compiler supports. The string is registered as a syntax template that has placeholders in it. These placeholders are supported: • `%n` — Replaced by the alignment boundary for the replacement function argument. • `%s` — Replaced by the aligned symbol, usually the identifier of a variable. For example, for the gcc compiler, you can specify `__attribute__((aligned(%n)))`, or for the MSVC compiler, `__declspec(align(%n))`.
`SupportedLanguage`	**Supported languages**	Cell array specifying the languages to which this alignment specification applies, among `c` and `c++`. Sometimes alignment syntax and position differ between languages for a compiler. .

Here is a data alignment specification for the GCC compiler:

```
da = RTW.DataAlignment;

as = RTW.AlignmentSpecification;
as.AlignmentType = {'DATA_ALIGNMENT_LOCAL_VAR', ...
                    'DATA_ALIGNMENT_STRUCT_FIELD', ...
                    'DATA_ALIGNMENT_GLOBAL_VAR'};
as.AlignmentSyntaxTemplate = '__attribute__((aligned(%n)))';
as.AlignmentPosition = 'DATA_ALIGNMENT_PREDIRECTIVE';
as.SupportedLanguages = {'c', 'c++'};
da.addAlignmentSpecification(as);

tc = RTW.TargetCharacteristics;
tc.DataAlignment = da;
```

Here is the corresponding specification in the **Generate customization** dialog box of the Code Replacement Tool.



Basic Example of Code Replacement Data Alignment

A simple example of the complete workflow for data alignment specified for code replacement is:

1. Create and save the following code replacement table definition file, `crl_table_mmul_4x4_single_align.m`. This table defines a replacement entry for the * (multiplication) operator, the `single` data type, and input dimensions `[4,4]`. The entry also specifies a data alignment boundary of 16 bytes for each replacement function argument. The entry expresses the requirement that the starting memory address for the data allocated for the function arguments during code generation is a multiple of 16.

```
function hLib = crl_table_mmul_4x4_single_align
%CRL_TABLE_MMUL_4x4_SINGLE_ALIGN - Describe matrix operator entry with data alignment

hLib = RTW.TflTable;
entry = RTW.TflCOperationEntry;
setTflCOperationEntryParameters(entry, ...
    'Key',                  'RTW_OP_MUL', ...
    'Priority',             90, ...
    'ImplementationName',   'matrix_mul_4x4_s');

% conceptual arguments
createAndAddConceptualArg(entry, 'RTW.TflArgMatrix',...
                                 'Name',         'y1', ...
                                 'IOType',       'RTW_IO_OUTPUT', ...
                                 'BaseType',     'single', ...
                                 'DimRange',     [4 4]);
```

```
createAndAddConceptualArg(entry, 'RTW.TflArgMatrix',...
                                 'Name',          'u1', ...
                                 'BaseType',      'single', ...
                                 'DimRange',      [4 4]);

createAndAddConceptualArg(entry, 'RTW.TflArgMatrix',...
                                 'Name',          'u2', ...
                                 'BaseType',      'single', ...
                                 'DimRange',      [4 4]);

% implementation arguments
arg = getTflArgFromString(hLib, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
entry.Implementation.setReturn(arg);

arg = getTflArgFromString(hLib, 'y1','single*');
desc = RTW.ArgumentDescriptor;
desc.AlignmentBoundary = 16;
arg.Descriptor = desc;
entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hlib, 'u1','single*');
desc = RTW.ArgumentDescriptor;
desc.AlignmentBoundary = 16;
arg.Descriptor = desc;
entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hLib, 'u2','single*');
desc = RTW.ArgumentDescriptor;
desc.AlignmentBoundary = 16;
arg.Descriptor = desc;
entry.Implementation.addArgument(arg);

hLib.addEntry(entry);
```

2 Create and save the following registration file, `rtwTargetInfo.m`. If you want to compile the code generated in this example, first modify the `AlignmentSyntaxTemplate` property for the compiler that you use. For example, for the MSVC compiler, replace the gcc template character vector `__attribute__((aligned(%n)))` with `__declspec(align(%n))`.

```
function rtwTargetInfo(cm)
% rtwTargetInfo function to register a code replacement library (CRL)
% for use with  code generation

  % Register the CRL defined in local function locCrlRegFcn
  cm.registerTargetInfo(@locCrlRegFcn);

end % End of RTWTARGETINFO

% Local function to define a CRL containing crl_table_mmul_4x4_single_align
function thisCrl = locCrlRegFcn

  % create an alignment specification object, assume gcc
```

```
    as = RTW.AlignmentSpecification;
    as.AlignmentType = {'DATA_ALIGNMENT_LOCAL_VAR', ...
                        'DATA_ALIGNMENT_GLOBAL_VAR', ...
                        'DATA_ALIGNMENT_STRUCT_FIELD'};
    as.AlignmentSyntaxTemplate = '__attribute__((aligned(%n)))';
    as.SupportedLanguages={'c', 'c++'};

    % add the alignment specification object
    da = RTW.DataAlignment;
    da.addAlignmentSpecification(as);

    % add the data alignment object to target characteristics
    tc = RTW.TargetCharacteristics;
    tc.DataAlignment = da;

    % Instantiate a CRL registry entry
    thisCrl = RTW.TflRegistry;

    % Define the CRL properties
    thisCrl.Name = 'Data Alignment Example';
    thisCrl.Description = 'Example of replacement with data alignment';
    thisCrl.TableList = {'crl_table_mmul_4x4_single_align'};
      thisCrl.TargetCharacteristics = tc;

end % End of LOCCRLREGFCN
```

3. To register your library with code generator without having to restart MATLAB, enter this command:

   ```
   RTW.TargetRegistry.getInstance('reset');
   ```

4. Configure the code generator to use your code replacement library.
5. Generate code and a code generation report.
6. Review the code replacements. For example, check whether a multiplication operation is replaced with a `matrix_mul_4x4_s` function call. In `mmalign.h`, check whether the gcc alignment directive `__attribute__((aligned(16)))` is generated to align the function variables.

More About

- “Code You Can Replace From Simulink Models” on page 51-7
- “Define Code Replacement Mappings” on page 51-42
- “Develop a Code Replacement Library” on page 51-27

Replace MATLAB Functions with Custom Code Using `coder.replace`

The `coder.replace` function provides the ability to replace a specified MATLAB function with a code replacement function in generated code. Use `coder.replace` in MATLAB code from which you want to generate C code using:

- MATLAB Coder
- MATLAB code in a Simulink MATLAB Function block

You can replace MATLAB functions that have:

- Single or multiple inputs
- Single or multiple outputs
- Scalar and matrix inputs and outputs

Supported types include:

- `single`, `double` (complex and noncomplex)
- `int8`, `uint8` (complex and noncomplex)
- `int16`, `uint16` (complex and noncomplex)
- `int32`, `uint32` (complex and noncomplex)
- Fixed-point integers
- Mixed types (different type on each input)

More About

- "Code You Can Replace From Simulink Models" on page 51-7
- "Define Code Replacement Mappings" on page 51-42
- "Develop a Code Replacement Library" on page 51-27

Replace `coder.ceval` Calls to External Functions

The `coder.ceval` function calls external C/C++ functions from code generated from MATLAB code. The code replacement software supports replacement of the function that you specify in a call to `coder.ceval`. An application of this code replacement scenario is to write generic MATLAB code that you can customize for different platforms with code replacements. A code replacement library can define hardware-specific code replacements for the function call. Use `coder.ceval` in MATLAB code from which you want to generate C code using:

- MATLAB Coder
- MATLAB code in a Simulink MATLAB Function block

Example Files

For the examples in "Interactive External Function Call Replacement Specification with Code Replacement Tool" on page 52-107 and "Programmatic External Function Call Replacement Specification" on page 52-108 you must have set up the following:

- Custom C function `my_add.c`.

```
/* my_add.c */

#include "my_add.h"

double my_add(double in1, double in2)
{
  return in1 + in2;
}
```

- Custom C header file `my_add.h`.

```
/* my_add.h */

double my_add(double in1, double in2);
```

- MATLAB function `call_my_add.m`, which uses `coder.ceval` to invoke `my_add.c`.

```
function y = call_my_add(in1, in2)  %#codegen

y=0.0;

if ~coder.target('Rtw')
% Executing in MATLAB, call MATLAB equivalent of C function my_add
```

```
  y= in1+in2;
else
% Executing in generated code, call C function my_add
  y = coder.ceval('my_add', in1, in2);
end
```

- MATLAB test function `call_my_add_test.m`, which calls `call_my_add.m`.

```
in1=10;
in2=20;

y = call_my_add(in1, in2);

disp('Output')
disp('y =')
disp(y);
```

- Replacement C function `my_add_replacement.c`.

```
/* my_add_replacement.c */

#include "my_add_replacement.h"

double my_add_replacement(double in1, double in2)
{
  return in1 + in2;
}
```

- Replacement C header file `my_add_replacement.h`.

```
/* my_add_replacement.h */

double my_add_replacement(double in1, double in2);
```

Interactive External Function Call Replacement Specification with Code Replacement Tool

This example shows how to define a code replacement table entry for a MATLAB function that calls `coder.ceval` to invoke an external C function. The example shows how to define the entry interactively with the Code Replacement Tool.

1 Identify or create the C/C++ code and relevant header files, the MATLAB function that calls `coder.ceval`, a MATLAB test function, and the source and header files for your replacement code. To follow along with this example, set up the files identified in “Example Files” on page 52-106.

2. In the Code Replacement Tool, add a table, select that table, and add a function entry. For more information, see "Define Code Replacement Mappings" on page 52-30.
3. On the **Mapping Information** tab, select `Custom` for the **Function** parameter.
4. In the **function-name** text box, type the custom function name. For this example, type the name `my_add`.
5. Under the **Conceptual arguments** list box, click **+** to add three arguments. By default, the tool creates an output argument `y1` and input arguments `u1` and `u2` of type `double`.
6. In the **Replacement function** > **Function prototype** section, type the name `my_add_replacement` in the **Name** text box.
7. Under the **Function arguments** list box, click **+** to add three function implementation arguments. By default, the tool creates an output argument `y1` and input arguments `u1` and `u2` of type double. Use the default settings.
8. In the **Function signature preview** box, if you see the expected function signature, click **Apply**. The function signature for this example, appears as:

   ```
   double my_add_replacement(double u1, double u2);
   ```
9. On the **Build Information** tab, specify `my_add_replacement.h` for the **Implementation header file** parameter and `my_add_replacement.c` for the **Implementation source file**.
10. Click **Validate entry**.
11. Save the code replacement table in the same folder as `my_add_replacement.c`. Name the file `crl_table_my_add.m`.

To test the example:

1. Register the table that contains the entry in a code replacement library.
2. Configure the code generator to use the code replacement library and to include the Code Replacements Report in the code generation report.
3. Generate code and the report.
4. Review the code replacements.

Programmatic External Function Call Replacement Specification

This example shows how to define a code replacement table entry for a MATLAB function that calls `coder.ceval` to invoke an external C function. The example shows how to define the entry programmatically.

1. Identify or create the C/C++ code and relevant header files, the MATLAB function that calls `coder.ceval` to invoke the C/C++ function, a MATLAB test function, and the source and header files for your replacement code. To follow along with this example, set up the files identified in "Example Files" on page 52-106.
2. Create a table definition file that contains a function definition. For example:

   ```
   function hLib = crl_table_my_add
   ```
3. Within the function body, create the table by calling the function `RTW.TflTable`.
4. Create an entry for the function mapping with a call to the `RTW.TflCFunctionEntry` function.

   ```
   hEnt = RTW.TflCFunctionEntry;
   ```
5. Set function entry parameters with a call to the `setTflCFunctionEntryParameters` function.

   ```
   hEnt.setTflCFunctionEntryParameters( ...
             'Key', 'my_add', ...
             'Priority', 100, ...
             'ImplementationName', 'my_add_replacement', ...
             'ImplementationHeaderFile', 'my_add_replacement.h', ...
             'ImplementationSourceFile', 'my_add_replacement.c');
   ```
6. Create conceptual arguments `y1`, `u1`, and `u1`. This example uses calls to the `getTflArgFromString` and `addConceptualArg` functions to create and add the arguments.

   ```
   arg = hEnt.getTflArgFromString('y1','double');
   arg.IOType = 'RTW_IO_OUTPUT';
   hEnt.addConceptualArg(arg);

   arg = hEnt.getTflArgFromString('u1','double');
   hEnt.addConceptualArg(arg);

   arg = hEnt.getTflArgFromString('u2','double');
   hEnt.addConceptualArg(arg);
   ```
7. Create the implementation arguments and add them to the entry. This example uses calls to the `getTflArgFromString` function to create implementation arguments. These functions map to arguments in the replacement function prototype: output argument `y1` and input arguments `u1` and `u2`. For each argument, the example uses the convenience method `setReturn` or `addArgument` to specify whether an argument is a return value or argument. For each argument, this example adds the argument to the entry array of implementation arguments.

```
arg = hEnt.getTflArgFromString('y1','double');
arg.IOType = 'RTW_IO_OUTPUT';
hEnt.Implementation.setReturn(arg);

arg = hEnt.getTflArgFromString('u1','double');
hEnt.Implementation.addArgument(arg);

arg = hEnt.getTflArgFromString('u2','double');
hEnt.Implementation.addArgument(arg);
```

8. Add the entry to a code replacement table with a call to the `addEntry` function.

   ```
   hLib.addEntry(hEnt);
   ```

9. Save the table definition file. Use the name of the table definition function to name the file.

To test the example:

1. Register the table that contains the entry in a code replacement library.
2. Configure the code generator to use the code replacement library and to include the Code Replacements Report in the code generation report.
3. Generate code and the report.
4. Review the code replacements.

More About

- “Code You Can Replace From Simulink Models” on page 51-7
- “Integrate MATLAB Algorithm in Model” (Simulink)
- “Define Code Replacement Mappings” on page 51-42
- “Develop a Code Replacement Library” on page 51-27

Replace MATLAB Functions Specified in MATLAB Function Blocks

This example shows how to use code replacement to replace a **MATLAB** function specified in a MATLAB Function block.

1. Open the `ex_replace` model. At the command prompt, enter:

   ```
   addpath(fullfile(docroot,'toolbox','ecoder','examples'))
   ex_replace
   ```

2. View the MATLAB Function Block code. In the model, double-click the MATLAB Function block to view the code in the MATLAB editor.

   ```
   function y = customFcn(u1, u2) %#codegen
   % This block supports MATLAB for code generation.

   % Replace this MATLAB function with CRL replacement function and if no
   % CRL replacement is found, generate an error during code generation.
   coder.replace('-errorifnoreplacement');

   assert(isa(u1,'int32'));
   assert(isa(u2,'int32'));

   y = power(u1,u2);
   ```

   The `coder.replace('-errorifnoreplacement')` statement instructs the code generator to replace this MATLAB function with a code replacement library function. The code generator produces an error if it does not find a match.

3. Create a table definition file that contains a function definition. For example:

   ```
   function hTable = crl_table_coderreplace()
   ```

4. Within the function body, create the table by calling the function `RTW.TflTable`.

   ```
   hTable = RTW.TflTable;
   ```

5. Create an entry for the function mapping with a call to the `RTW.TflCFunctionEntry` function.

   ```
   hEnt = RTW.TflCFunctionEntry;
   ```

6. Set function entry parameters with a call to the `setTflCFunctionEntryParameters` function.

   ```
   setTflCFunctionEntryParameters(hEnt, ...
       'Key',                    'customFcn', ...
   ```

```
    'Priority',                100, ...
    'ImplementationName',      'scalarFcnReplacement', ...
    'ImplementationHeaderFile', 'MyMath.h', ...
    'ImplementationSourceFile', 'MyMath.c')
```

7 Create conceptual arguments `y1`, `u1`, and `u1`. This example uses calls to the `getTflArgFromString` and `addConceptualArg` functions to create and add the arguments.

```
arg = getTflArgFromString(hEnt, 'y1','int32');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEnt, arg);

arg = getTflArgFromString(hEnt, 'u1','int32');
addConceptualArg(hEnt, arg);

arg = getTflArgFromString(hEnt, 'u2','int32');
addConceptualArg(hent, arg);
```

8 Create the implementation arguments and add them to the entry. This example uses calls to the `getTflArgFromString` function to create implementation arguments that map to arguments in the replacement function prototype: output argument *`void`*, input arguments *`u1`* and *`u2`*, and output argument *`y1`*. The convenience methods `setReturn` and `addArgument` specify whether an argument is a return value or argument. The `addArgument` function also adds each argument to the entry's array of implementation arguments.

```
arg = getTflArgFromString(hEnt, 'void','void');
arg.IOType = 'RTW_IO_OUTPUT';
hEnt.Implementation.setReturn(arg);

arg = getTflArgFromString(hEnt, 'u1','int32');
hEnt.Implementation.addArgument(arg);

arg = getTflArgFromString(hEnt, 'u2','int32');
hEnt.Implementation.addArgument(arg);

arg = getTflArgFromString(hEnt, 'y1','int32*');
arg.IOType = 'RTW_IO_OUTPUT';
hEnt.Implementation.addArgument(arg);
```

9 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hLib, hEnt);
```

10 Save the table definition file. Use the name of the table definition function to name the file.

To test the example:

1. Register the code replacement mapping.
2. Create files `MyMath.c` and `MyMath.h` that define the replacement function, `scalarFcnReplacement`, which has two `int32` inputs and one `int32` output.

   `MyMath.c`

   ```
   #include "MyMath.h"

   void scalarFcnReplacement(int32_T u1, int32_T u2, int32_T* y1 ) {
       *y1 = u1^u2;
   }
   ```

   `MyMath.h`

   ```
   #ifndef _ScalarMath_h
   #define _ScalarMath_h

   #include "rtwtypes.h"

   #ifdef __cplusplus
   extern "C" {
   #endif

   extern void scalarFcnReplacement(int32_T u1, int32_T u2, int32_T* y1);

   #ifdef __cplusplus
   }
   #endif

   #endif
   ```
3. Open the `ex_replace` model.
4. Configure the code generator to use the code replacement library and to include the Code Replacements Report in the code generation report.
5. Generate the replacement code and a code generation report.
6. Review the code replacements. In the code generation report, view the generated code for `ex_replace.c`.

   ```
   void ex_replace_step(void)
   {
      int32_T y;
      scalarFcnReplacement(ex_replace_U.In1, ex_replace_U.In2, &y);
      ex_replace_Y.Out1 = y;
   ```

```
}
```

More About

- “Code You Can Replace From Simulink Models” on page 51-7
- “Define Code Replacement Mappings” on page 51-42
- “Develop a Code Replacement Library” on page 51-27

Reserved Identifiers and Code Replacement

The code generator and C programming language use, internally, reserved keywords for code generation. Do not use reserved keywords as identifiers or function names. Reserved keywords for code generation include many code replacement library identifiers, the majority of which are function names, such as `acos`.

To view a list of reserved identifiers for the code replacement library that you use to generate code, specify the name of the library in a call to the function `RTW.TargetRegistry.getInstance.getTflReservedIdentifiers`. For example:

```
crl_ids = RTW.TargetRegistry.getInstance.getTflReservedIdentifiers('GNU99 (GNU)')
```

In a code replacement table, the code generator registers each function implementation name defined by a table entry 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 code replacement reserved identifiers, use the `setReservedIdentifiers` function. This function registers specified reserved identifiers to be associated with a code replacement table.

You can register up to four reserved identifier structures in a code replacement table. You can associate one set of reserved identifiers with a code replacement library, while the other three (if present) must be associated with ANSI C. The following example shows a reserved identifier structure that specifies two identifiers and the associated header file.

```
d{1}.LibraryName = 'ANSI_C';
d{1}.HeaderInfos{1}.HeaderName = 'math.h';
d{1}.HeaderInfos{1}.ReservedIds = {'y0', 'y1'};
```

The code generator adds the identifiers to the list of reserved identifiers and honors them during the build procedure.

More About

- “Code You Can Replace From Simulink Models” on page 51-7
- “Customize Match and Replacement Process” on page 51-153
- “Define Code Replacement Mappings” on page 51-42
- “Develop a Code Replacement Library” on page 51-27

Customize Match and Replacement Process

During the build process, the code generator uses:

- Preset match criteria to identify functions and operators for which application-specific implementations replace default implementations.
- Preset replacement function signatures.

It is possible that preset match criteria and preset replacement function signatures do not completely meet your function and operator replacement needs. For example:

- You want to replace an operator with a particular fixed-point implementation function only when fraction lengths are within a particular range.
- When a match occurs, you want to modify your replacement function signature based on compile-time information, such as passing fraction-length values into the function.

To add extra logic into the code replacement match and replacement process, create custom code replacement table entries. With custom entries, you can specify additional match criteria and modify the replacement function signature to meet application needs.

To create a custom code replacement entry:

1. Create a custom code replacement entry class, derived from `RTW.TflCFunctionEntryML (for function replacement)` or `RTW.TflCOperationEntryML` (for operator replacement).
2. In your derived class, implement a `do_match` method with a fixed preset signature as a MATLAB function. In your `do_match` method, provide either or both of the following customizations that instantiate the class:

    - Add match criteria that the base class does not provide. The base class provides a match based on:

        - Argument number
        - Argument name
        - Signedness
        - Word size
        - Slope (if not specified with wildcards)
        - Bias (if not specified with wildcards)
        - Math modes, such as saturation and rounding

    - Operator or function key
  - Modify the implementation signature by adding additional arguments or setting constant input argument values. You can inject a constant value, such as an input scaling value, as an additional argument to the replacement function.
3. Create code replacement entries that instantiate the custom entry class.
4. Register a library containing the code replacement table that includes your entries.

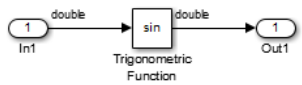
During code generation, the code replacement match process tries to match function or operator call sites with the base class of your derived entry class. If the process finds a match, the software calls your `do_match` method to execute your additional match logic (if any) and your replacement function customizations (if any).

Customize Code Match and Replacement for Functions

This example shows how to use custom code replacement table entries to refine the match and replacement logic for functions. The example shows how to:

- Modify a sine function replacement only if the integer size on the current target platform is 32 bits.
- Change the replacement such that the implementation function passes in a degrees-versus-radians flag as an input argument.

1. To exercise the table entries that you create in this example, create an ERT-based model with a sine function block. For example:



   In the Inport block parameters, set the signal **Data type** to `double`. If the value selected for **Configuration Parameters** > **Hardware Implementation** > **Device type** supports an integer size other than 32, do one of the following:

   - Select a temporary target platform with a 32-bit integer size.
   - Modify the code to match the integer size of your target platform.
2. Create a class, for example `TflCustomFunctionEntry`, that is derived from the base class `RTW.TflCFunctionEntryML`. The derived class defines a `do_match` method with the signature:

```
function ent = do_match(hThis, ...
        hCSO, ...
        targetBitPerChar, ...
        targetBitPerShort, ...
        targetBitPerInt, ...
        targetBitPerLong, ...
        targetBitPerLongLong)
```

In the `do_match` signature:

- `ent` is the return handle, which is returned either as empty (indicating that the match failed) or as a `TflCFunctionEntry` handle.
- `hThis` is a handle to the class instance.
- `hCSO` is a handle to an object that the code generator creates for querying the library for a replacement.
- Remaining arguments are the number of bits for various data types of the current target.

The `do_match` method:

- Adds required additional match criteria that the base class does not provide.
- Makes required modifications to the implementation signature.

In this case, the `do_match` method must match only `targetBitPerInt`, representing the number of bits in the C `int` data type for the current target, to the value 32. If the code generator finds a match, the method sets the return handle and creates and adds an input argument. The input argument represents whether units are expressed as degrees or radians, to the replacement function signature.

Alternatively, create and add the additional implementation function argument for passing a units flag in each code replacement table definition file that instantiates this class. In that case, this class definition code does not create the argument. That code sets only the argument value. For an example of creating and adding additional implementation function arguments in a table definition file, see “Customize Code Match and Replacement for Scalar Operations” on page 51-161.

```
classdef TflCustomFunctionEntry < RTW.TflCFunctionEntryML
  methods
    function ent = do_match(hThis, ...
        hCSO, ... %#ok
        targetBitPerChar, ... %#ok
```

```
        targetBitPerShort, ... %#ok
        targetBitPerInt, ... %#ok
        targetBitPerLong, ... %#ok
        targetBitPerLongLong) %#ok
      % DO_MATCH - Create a custom match function. The base class
      % checks the types of the arguments prior to calling this
      % method. This will check additional data and perhaps modify
      % the implementation function.

      ent = []; % default the return to empty, indicating the match failed.

      % Match sine function only if the target int size is 32 bits
      if targetBitPerInt == 32
        % Need to modify the default implementation, starting from a copy
        % of the standard TflCFunctionEntry.
        ent = RTW.TflCFunctionEntry(hThis);

        % If the target int size is 32 bits, the implementation function
        % takes an additional input flag argument indicating degress vs.
        % radians. The additional argument can be created and added either
        % in the CRL table definition file that instantiates this class, or
        % here in the class definition, as follows:
        createAndAddImplementationArg(ent, 'RTW.TflArgNumericConstant', ...
                                      'Name',           'u2', ...
                                      'IsSigned',       true, ...
                                      'WordLength',     32, ...
                                      'FractionLength', 0, ...
                                      'Value',          1);
      end
    end
  end
end
```

Exit the class folder and return to the previous working folder.

3 Create and save the following code replacement table definition file, `crl_table_custom_sinfcn_double.m`. This file defines a code replacement table that contains a function table entry for sine with `double` input and output. This entry instantiates the derived class from the previous step, `TflCustomFunctionEntry`.

```
function hTable = crl_table_custom_sinfcn_double

hTable = RTW.TflTable;

%% Add TflCustomFunctionEntry
fcn_entry = TflCustomFunctionEntry;
setTflCFunctionEntryParameters(fcn_entry, ...
    'Key',                    'sin', ...
    'Priority',               30, ...
    'ImplementationName',     'mySin', ...
    'ImplementationHeaderFile', 'mySin.h', ...
    'ImplementationSourceFile', 'mySin.c');
```

```
createAndAddConceptualArg(fcn_entry, 'RTW.TflArgNumeric', ...
     'Name',          'y1', ...
     'IOType',        'RTW_IO_OUTPUT', ...
     'DataTypeMode', 'double');

createAndAddConceptualArg(fcn_entry, 'RTW.TflArgNumeric', ...
     'Name',          'u1', ...
     'IOType',        'RTW_IO_INPUT', ...
     'DataTypeMode', 'double');

% TflCustomFunctionEntry class do_match method will create and add
% an implementation function argument during code generation if
% the supported integer size on the current target is 32 bits.
copyConceptualArgsToImplementation(fcn_entry);

addEntry(hTable, fcn_entry);
```

4 Check the validity of the code replacement table entry.

- At the command prompt, invoke the table definition file.

  ```
  tbl = crl_table_custom_sinfcn_double
  ```

- In the Code Replacement Viewer, view the table definition file.

  ```
  crviewer(crl_table_custom_sinfcn_double)
  ```

Customize Code Match and Replacement for Nonscalar Operations

This example shows how to create custom code replacement entries that add logic to the code match and replacement process for a nonscalar operation. Custom entries specify additional match criteria or modify the replacement function signature to meet application needs.

This example restricts the match criteria for an element-wise multiplication replacement to entries with a specific dimension range. When a match occurs, the custom `do_match` method modifies the replacement signature to pass the number of elements into the function.

Files for developing and testing this code replacement library example are available in `matlab/help/toolbox/ecoder/examples/code_replacement/custom_elemmult`:

- `do_match` method — `@MyElemMultEntry/MyElemMultEntry.m`
- Replacement function source and header files — `src/myMulImplLib.c` and `src/myMulImplLib.h`
- Model — `myElemMul.slx`

- Code replacement table definition — `myElemMultCrlTable.m`
- Registration file — `rtwTargetInfo.m`

To create custom code replacement entries that add logic to the code replacement match and replacement process:

1. Create a class, for example `MyElemMultEntry`, which is derived from the base class `RTW.TflCOperationEntryML`. The derived class defines a `do_match` method with the following signature:

   ```
   function ent = do_match(hThis, ...
       hCSO, ...
       targetBitPerChar, ...
       targetBitPerShort, ...
       targetBitPerInt, ...
       targetBitPerLong, ...
       targetBitPerLongLong)
   ```

   In the `do_match` signature:

   - `ent` is the return handle, which is returned as empty (indicating that the match failed) or as a `TflCOperationEntry` handle.
   - `hThis` is the handle to the derived instance.
   - `hCSO` is a handle to an object that the code generator creates for querying the library for a replacement.
   - Remaining arguments are the number of bits for various data types of the current target.

   The `do_match` method:

   - Adds match criteria that the base class does not provide.
   - Makes changes to the implementation signature.

   The `do_match` method relies on the base class for checking data types and dimension ranges. If the code generator finds a match, `do_match`:

   - Sets the return handle.
   - Uses the conceptual arguments to compute the number of elements in the array. In the replacement entry returned, sets the value of the constant implementation argument as the number of elements of the array.

- Updates the code replacement entry such that it matches CSOs that have the same argument dimensions.

```
classdef MyElemMulyEntry < RTW.TflCOperationEntryML
    methods
        function obj = MyElemMultEntry(varargin)
            mlock;
            obj@RTW.TflCOperationEntryML(varargin{:});
        end

        function ent = do_match(hThis, ...
            hCSO, ... %#ok
            targetBitPerChar, ... %#ok
            targetBitPerShort, ... %#ok
            targetBitPerInt, ... %#ok
            targetBitPerLong, ... %#ok
            targetBitPerLongLong ) %#ok

            %  Fourth implementation arg represents number of elements for producing matches.
            assert(strcmp(hThis.Implementation.Arguments(4).Name,'numElements'));

            ent = RTW.TflCOperationEntry(hThis);

            % Calculate number of elements and set value of injected constant.
            ent.Implementation.Arguments(4).Value = prod(hCSO.ConceptualArgs(1).DimRange(1,:));

            % Since implementation has been modified for specific DimRange, update
            % returned entry to match similar CSOs only.
            for idx =1:3
                ent.ConceptualArgs(idx).DimRange = hCSO.ConceptualArgs(idx).DimRange;
            end
        end
    end
end
```

2 Create and save the following code replacement table definition file, `myElemMultCrlTable.m`. This file defines a code replacement table that contains an operator entry generator for element-wise multiplication. The table entry:

- Instantiates the derived class `myElemMultEntry` from the previous step.
- Sets operator entry parameters with the call to the `setTflCOperationEntryParameters` function.
- Creates conceptual arguments `y1`, `u1`, and `u2`. The argument class `RTW.TflArgMatrix` specifies matrix arguments to match. The three arguments are set up to match 2-dimensional matrices with at least two elements in each dimension.

- Calls the `getTflArgFromString` function to create a return value and four implementation arguments. Arguments `u1` and `u2` are the operands, `y1` is the product, and the fourth argument is the number of elements.

  Alternatively, the `do_match` method of the derived class `myElemMultEntry` can create and add the implementation arguments. When the number of additional implementation arguments required can vary based on compile-time information, use the alternative approach.
- Calls `addEntry` to add the entry to a code replacement table.

```
function hLib = myElemMultCrlTable

libPath = fullfile(fileparts(which(mfilename)),'src');

hLib = RTW.TflTable;
%---------- entry: RTW_OP_ELEM_MUL -----------
hEnt = MyElemMultEntry;
hEnt.setTflCOperationEntryParameters( ...
          'Key', 'RTW_OP_ELEM_MUL', ...
          'Priority', 100, ...
          'SaturationMode', 'RTW_WRAP_ON_OVERFLOW', ...
          'ImplementationName', 'myElemMul_s32', ...
          'ImplementationSourceFile', 'myMulImplLib.c', ...
          'ImplementationSourcePath', libPath, ...
          'ImplementationHeaderFile', 'myMulImplLib.h', ...
          'ImplementationHeaderPath', libPath, ...
          'SideEffects', true, ...
          'GenCallback','RTW.copyFileToBuildDir');

% Conceptual Args

arg = RTW.TflArgMatrix('y1', 'RTW_IO_OUTPUT',  'int32');
arg.DimRange = [2 2; Inf Inf];
hEnt.addConceptualArg(arg);

arg = RTW.TflArgMatrix('u1', 'RTW_IO_INPUT',  'int32');
arg.DimRange = [2 2; Inf Inf];
hEnt.addConceptualArg(arg);

arg = RTW.TflArgMatrix('u2', 'RTW_IO_INPUT',  'int32');
arg.DimRange = [2 2; Inf Inf];
hEnt.addConceptualArg(arg);
```

```
% Implementation Args

arg = hEnt.getTflArgFromString('unused','void');
arg.IOType = 'RTW_IO_OUTPUT';
hEnt.Implementation.setReturn(arg);

arg = hEnt.getTflArgFromString('u1','int32*');
hEnt.Implementation.addArgument(arg);

arg = hEnt.getTflArgFromString('u2','int32*');
hEnt.Implementation.addArgument(arg);

arg = hEnt.getTflArgFromString('y1','int32*');
arg.IOType = 'RTW_IO_OUTPUT';
hEnt.Implementation.addArgument(arg);

arg = hEnt.getTflArgFromString('numElements','uint32',0);
hEnt.Implementation.addArgument(arg);

hLib.addEntry( hEnt );
```

3 Check the validity of the code replacement table entry.

- At the command prompt, invoke the table definition file.

  ```
  tbl = myElemMultCrlTable
  ```

- In the Code Replacement Viewer, view the table definition file.

  ```
  crviewer(myElemMultCrlTable)
  ```

Customize Code Match and Replacement for Scalar Operations

This example shows how to create custom code replacement entries that add logic to the code match and replacement process for a scalar operation. Custom entries specify additional match criteria or modify the replacement function signature to meet application needs.

For example:

- When fraction lengths are within a specific range, replace an operator with a fixed-point implementation function.
- When a match occurs, modify the replacement function signature based on compile-time information, such as passing fraction-length values into the function.

This example modifies a fixed-point addition replacement such that the implementation function passes in the fraction lengths of the input and output data types as arguments.

To create custom code replacement entries that add logic to the code replacement match and replacement process:

1 Create a class, for example `TflCustomOperationEntry`, that is derived from the base class `RTW.TflCOperationEntryML`. The derived class defines a `do_match` method with the following signature:

```
function ent = do_match(hThis, ...
        hCSO, ...
        targetBitPerChar, ...
        targetBitPerShort, ...
        targetBitPerInt, ...
        targetBitPerLong, ...
        targetBitPerLongLong)
```

In the `do_match` signature:

- `ent` is the return handle, which is returned as empty (indicating that the match failed) or as a `TflCOperationEntry` handle.
- `hThis` is the handle to the class instance.
- `hCSO` is a handle to an object that the code generator creates for querying the library for a replacement.
- Remaining arguments are the number of bits for various data types of the current target.

The `do_match` method adds match criteria that the base class does not provide. The method makes modifications to the implementation signature. In this case, the `do_match` method relies on the base class for checking word size and signedness. `do_match` must match only the number of conceptual arguments to the value 3 (two inputs and one output) and the bias for each argument to value 0. If the code generator finds a match, `do_match`:

- Sets the return handle.
- Removes slope and bias wild cards from the conceptual arguments (the match is for specific slope and bias values).
- Writes fraction-length values for the inputs and output into replacement function arguments 3, 4, and 5.

You can create and add three additional implementation function arguments for passing fraction lengths in the class definition or in each code replacement entry definition that instantiates this class. This example creates the arguments, adds them to a code replacement table definition file, and sets them to specific values in the class definition code.

```
classdef TflCustomOperationEntry < RTW.TflCOperationEntryML
  methods
    function ent = do_match(hThis, ...
        hCSO, ... %#ok
        targetBitPerChar, ... %#ok
        targetBitPerShort, ... %#ok
        targetBitPerInt, ... %#ok
        targetBitPerLong, ... %#ok
        targetBitPerLongLong) %#ok

      % DO_MATCH - Create a custom match function. The base class
      % checks the types of the arguments prior to calling this
      % method. This class will check additional data and can
      % modify the implementation function.

      % The base class checks word size and signedness. Slopes and biases
      % have been wildcarded, so the only additional checking to do is
      % to check that the biases are zero and that there are only three
      % conceptual arguments (one output, two inputs)

      ent = []; % default the return to empty, indicating the match failed

      if length(hCSO.ConceptualArgs) == 3 && ...
          hCSO.ConceptualArgs(1).Type.Bias == 0 && ...
          hCSO.ConceptualArgs(2).Type.Bias == 0 && ...
          hCSO.ConceptualArgs(3).Type.Bias == 0

        % Modify the default implementation. Since this is a
        % generator entry, a concrete entry is created using this entry
        % as a template. The type of entry being created is a standard
        % TflCOperationEntry. Using the standard operation entry
        % provides required information, and you do not need
        % a custom match function.
        ent = RTW.TflCOperationEntry(hThis);

        % Since this entry is modifying the implementation for specific
        % fraction-length values (arguments 3, 4, and 5), the conceptual argument
        % wild cards must be removed (the wildcards were inherited from the
        % generator when it was used as a template for the concrete entry).
        % This concrete entry is now for a specific slope and bias.
        % hCSO holds the slope and bias values (created by the code generator).
        for idx=1:3
          ent.ConceptualArgs(idx).CheckSlope = true;
          ent.ConceptualArgs(idx).CheckBias = true;

          % Set the specific Slope and Biases
```

```
            ent.ConceptualArgs(idx).Type.Slope = hCSO.ConceptualArgs(idx).Type.Slope;
            ent.ConceptualArgs(idx).Type.Bias = 0;
          end

          % Set the fraction-length values in the implementation function.
          ent.Implementation.Arguments(3).Value = ...
             -1.0*hCSO.ConceptualArgs(2).Type.FixedExponent;
          ent.Implementation.Arguments(4).Value = ...
             -1.0*hCSO.ConceptualArgs(3).Type.FixedExponent;
          ent.Implementation.Arguments(5).Value = ...
             -1.0*hCSO.ConceptualArgs(1).Type.FixedExponent;
        end
      end
    end
  end
```

Exit the class folder and return to the previous working folder.

2 Create and save the following code replacement table definition file, `crl_table_custom_sinfcn_double.m`. This file defines a code replacement table that contains a single operator entry, an entry generator for unsigned 32-bit fixed-point addition operations, with arbitrary fraction-length values on the inputs and the output. The table entry:

- Instantiates the derived class `TflCustomOperationEntry` from the previous step. If you want to replace word sizes and signedness attributes, you can use the same derived class, but not the same entry, because you cannot use a wild card with the `WordLength` and `IsSigned` arguments. For example, to support `uint8`, `int8`, `uint16`, `int16`, and `int32`, add five other distinct entries. To use different implementation functions for saturation and rounding modes other than overflow and round to floor, add entries for those match permutations.
- Sets operator entry parameters with the call to the `setTflCOperationEntryParameters` function.
- Calls the `createAndAddConceptualArg` function to create conceptual arguments `y1`, `u1`, and `u2`.
- Calls `createAndSetCImplementationReturn` and `createAndAddImplementationArg` to define the signature for the replacement function. Three of the calls to `createAndAddImplementationArg` create implementation arguments to hold the fraction-length values for the inputs and output. Alternatively, the entry can omit those argument definitions. Instead, the `do_match` method of the derived class `TflCustomOperationEntry` can create and add the three implementation arguments. When the number of additional implementation arguments required can vary based on compile-time information, use the alternative approach.

- Calls `addEntry` to add the entry to a code replacement table.

```
function hTable = crl_table_custom_add_ufix32

hTable = RTW.TflTable;

%% Add TflCustomOperationEntry
op_entry = TflCustomOperationEntry;

setTflCOperationEntryParameters(op_entry, ...
    'Key',                      'RTW_OP_ADD', ...
    'Priority',                 30, ...
    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes',            {'RTW_ROUND_FLOOR'}, ...
    'ImplementationName',       'myFixptAdd', ...
    'ImplementationHeaderFile', 'myFixptAdd.h', ...
    'ImplementationSourceFile', 'myFixptAdd.c');

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
     'Name',       'y1', ...
     'IOType',     'RTW_IO_OUTPUT', ...
     'CheckSlope', false, ...
     'CheckBias',  false, ...
     'DataType',   'Fixed', ...
     'Scaling',    'BinaryPoint', ...
     'IsSigned',   false, ...
     'WordLength', 32);

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
      'Name',       'u1', ...
      'IOType',     'RTW_IO_INPUT', ...
      'CheckSlope', false, ...
      'CheckBias',  false, ...
      'DataType',   'Fixed', ...
      'Scaling',    'BinaryPoint', ...
      'IsSigned',   false, ...
      'WordLength', 32);

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
       'Name',       'u2', ...
       'IOType',     'RTW_IO_INPUT', ...
       'CheckSlope', false, ...
       'CheckBias',  false, ...
       'DataType',   'Fixed', ...
```

```
        'Scaling',    'BinaryPoint', ...
        'IsSigned',   false, ...
        'WordLength', 32);

% Specify replacement function signature
createAndSetCImplementationReturn(op_entry, 'RTW.TflArgNumeric', ...
    'Name',         'y1', ...
    'IOType',       'RTW_IO_OUTPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric', ...
    'Name',         'u1', ...
    'IOType',       'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric', ...
    'Name',         'u2', ...
    'IOType',       'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0);

% Add 3 fraction-length args. Actual values are set during code generation.
createAndAddImplementationArg(op_entry, 'RTW.TflArgNumericConstant', ...
    'Name',         'fl_in1', ...
    'IOType',       'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value',         0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumericConstant', ...
    'Name',         'fl_in2', ...
    'IOType',       'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value',         0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumericConstant', ...
```

```
    'Name',        'fl_out', ...
    'IOType',      'RTW_IO_INPUT', ...
    'IsSigned',    false, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value',        0);

addEntry(hTable, op_entry);
```

3 Check the validity of the operator entry.

- At the command prompt, invoke the table definition file.

  ```
  tbl = crl_table_custom_sinfcn_double
  ```

- In the Code Replacement Viewer, view the table definition file.

  ```
  crviewer(crl_table_custom_sinfcn_double)
  ```

More About

- "Code You Can Replace From Simulink Models" on page 51-7
- "Define Code Replacement Mappings" on page 51-42
- "Develop a Code Replacement Library" on page 51-27

Scalar Operator Code Replacement

This example shows how to define a code replacement mapping for a scalar operator. The example defines a mapping for the + (addition) operator programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

1. Create a table definition file that contains a function definition. For example:

   ```
   function hTable = crl_table_add_uint8
   ```

2. Within the function body, create the table by calling the function `RTW.TflTable`.

   ```
   hTable = RTW.TflTable;
   ```

3. Create an entry for the operator mapping with a call to the `RTW.TflCOperationEntry` function.

   ```
   % Create operation entry
   op_entry = RTW.TflCOperationEntry;
   ```

4. Set function entry parameters with a call to the `setTflCOperationEntryParameters` function.

   ```
   % Define addition operation of built-in uint8 data type
   % Saturation on, Rounding unspecified
   setTflCOperationEntryParameters(op_entry, ...
                       'Key',                      'RTW_OP_ADD', ...
                       'Priority',                 90, ...
                       'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
                       'RoundingModes',            {'RTW_ROUND_UNSPECIFIED'}, ...
                       'ImplementationName',       'u8_add_u8_u8', ...
                       'ImplementationHeaderFile', 'u8_add_u8_u8.h', ...
                       'ImplementationSourceFile', 'u8_add_u8_u8.c' );
   ```

5. Create conceptual arguments `y1`, `u1`, and `u2`. There are multiple ways to set up the conceptual arguments. This example uses calls to the `getTflArgFromString` and `addConceptualArg` functions to create and add the arguments.

   ```
   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 );
   ```

6. Copy the conceptual arguments to the implementation arguments. There are multiple ways to set up the implementation arguments. This example uses a call

to the `copyConceptualArgsToImplementation` function to create and add implementation arguments to the entry by copying matching conceptual arguments.

```
copyConceptualArgsToImplementation(op_entry);
```

7 Add the entry to a code replacement table with a call to the `addEntry` function.

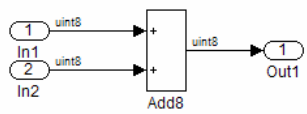
```
addEntry(hTable, op_entry);
```

8 Save the table definition file. Use the name of the table definition function to name the file.

To test this example:

1 Register the code replacement mapping.

2 Create a model that includes an Add block, such as this model.



3 Configure the model with the following settings:

- On the **Solver** pane, select a fixed-step solver.
- On the **Code Generation** pane, select an ERT-based system target file.
- On the **Code Generation** > **Interface** pane, select the code replacement library that contains your addition operation entry.

4 Generate code and a code generation report.

5 Review the code replacements.

More About

- "Code You Can Replace From Simulink Models" on page 51-7
- "Define Code Replacement Mappings" on page 51-42
- "Data Alignment for Code Replacement" on page 51-133
- "Remap Operator Output to Function Input" on page 51-192
- "Customize Match and Replacement Process" on page 51-153
- "Develop a Code Replacement Library" on page 51-27
- "What Is Code Replacement Customization?" on page 51-3

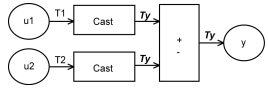
Addition and Subtraction Operator Code Replacement

Consider the following when defining mappings for addition and subtraction operator code replacements.

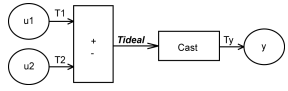
Algorithm Options

When creating a code replacement table entry for an addition or subtraction operator, first determine the type of algorithm that your library function implements.

- Cast-before-operation (CBO), default — Prior to performing the addition or subtraction operation, the algorithm type casts input values to the output type. If the output data type cannot exactly represent the input values, losses can occur as a result of the cast to the output type. Additional loss can occur when the result of the operation is cast to the final output type.



- Cast-after-operation (CAO) — The algorithm computes the ideal result of the addition or subtraction operation of the two inputs. The algorithm then type casts the result to the output data type. Loss occurs during the type cast. This algorithm behaves similarly to the C language except when the signedness of the operands does not match. For example, when you add a signed long operand to an unsigned long operand, standard C language rules convert the signed long operand to an unsigned long operand. The result is a value that is not ideal.



Interactive Specification with Code Replacement Tool

When you use the Code Replacement Tool to create a code replacement table entry for an addition or subtraction operation, the tool displays an **Algorithm** menu. Use that menu to specify the `Cast before operation` or `Cast after operation` algorithm for that entry.

Programmatic Specification

Create a code replacement table file, as a MATLAB function, that describes the addition or subtraction code replacement table entry. In the call to `setTflCOperationEntryParameters`, set at least these parameters:

- `Key` to `RTW_OP_ADD` or `RTW_OP_MINUS`
- `ImplementationName` to the name of your replacement function
- `EntryInfoAlgorithm` to `RTW_CAST_BFORE_OP` (cast-before-operation) or `RTW_CAST_AFTER_OP` (cast-after-operation)

This example sets parameters for a code replacement operator entry for a cast-after-operation implementation of a `uint8` addition.

```
op_entry = RTW.TflCOperationEntry;
setTflCOperationEntryParameters(op_entry, ...
                  'Key',                      'RTW_OP_ADD', ...
                  'EntryInfoAlgorithm',       'RTW_CAST_AFTER_OP', ...
                  'ImplementationName',       'u8_add_u8_u8');
```

For more information, see `setTflCOperationEntryParameters`.

Algorithm Classification

During code generation, the code generator examines addition and subtraction operations, including adjacent type cast operations, to determine the type of algorithm to compute the expression result. Based on the data types in the expression and the type of the accumulator (type used to hold the result of the addition or subtraction operation), the code generator uses these rules.

- Floating-point types only

Input 1 Data Type	Input 2 Data Type	Accumulator Data Type	Output Data Type	Classification
`double`	`double`	`double`	`double`	CBO, CAO
`double`	`double`	`double`	`single`	—
`double`	`double`	`single`	`double`	—
`double`	`double`	`single`	`single`	CBO
`double`	`single`	`double`	`double`	CBO, CAO

Input 1 Data Type	Input 2 Data Type	Accumulator Data Type	Output Data Type	Classification
`double`	`single`	`double`	`single`	—
`double`	`single`	`single`	`double`	—
`double`	`single`	`single`	`single`	CBO
`single`	`single`	`single`	`single`	CBO, CAO
`single`	`single`	`single`	`double`	—
`single`	`single`	`double`	`single`	—
`single`	`single`	`double`	`double`	CBO, CAO

- Floating-point and fixed-point types on the immediate addition or subtraction operation

Algorithm	Conditions
CBO	One of the following is true: • Operation type is `double`. • Operation type is `single` and input types are `single` or fixed-point.
CAO	Operation type is a superset of input types—that is, output type can represent values of input types without loss of data.

- Fixed-point types only

Algorithm	Conditions
CBO	At least one of the following is true: • Accumulator type equals output type (`Tacc == Tout`). • Output type is a superset of input types (`Tacc >= {Tin1, Tin2}`) and accumulator type is a superset of output type (`Tacc >= Tout`). • Operation does not incur range or precision loss.
CAO	Net bias is zero and the data types in the expression have equal slope adjustment factors. For more information on net bias, see "Addition" or "Subtraction" in "Fixed-Point Operator Code Replacement" on page 52-146 (for MATLAB code) or "Fixed-Point Operator Code Replacement" on page 51-195 (for Simulink models).

In many cases, the numerical result of a CBO operation is equal to that of a CAO operation. For example, if the input and output types are such that the operation produces the ideal result, as in the case of `int8 + int8 –> int16`. To maximize the probability of code replacement occurring in such cases, set the algorithm to cast-after-operation.

Limitations

- The code generator does not replace operations with nonzero net bias.
- When classifying an operation as a CAO operation, the code generator includes the adjacent casts in the expression when the expression involves only fixed-point types. Otherwise, the code generator classifies and replaces only the immediate addition or subtraction operation. Casts that the code generator excludes from the classification appear in the generated code.
- To enable the code generator to include multiple cast operations, which follow an addition or subtraction of fixed-point data, in the classification of an expression, the rounding mode must be `simplest` or `floor`. Consider the expression `y=(cast A)(cast B)(u1+u2)`. If the rounding mode of `(cast A)`, `(cast B)`, and the addition operator (+) are set to `simplest` or `floor`, the code generator takes into account `(cast A)` and `(cast B)` when classifying the expression and performing the replacement only.

More About

- "Code You Can Replace From Simulink Models" on page 51-7
- "Define Code Replacement Mappings" on page 51-42
- "Data Alignment for Code Replacement" on page 51-133
- "Remap Operator Output to Function Input" on page 51-192
- "Customize Match and Replacement Process" on page 51-153
- "Fixed-Point Operator Code Replacement" on page 51-195
- "Develop a Code Replacement Library" on page 51-27

Small Matrix Operation to Processor Code Replacement

This example shows how to define code replacement mappings that replace nonscalar small matrix operations with processor-specific intrinsic functions. The example defines a table containing two matrix operator replacement entries for the + (addition) operator and the `double` data type. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

1. Create a table definition file that contains a function definition. For example:

   ```
   function hTable = crl_table_matrix_add_double
   ```

2. Within the function body, create the table by calling the function `RTW.TflTable`.

   ```
   hTable = RTW.TflTable;
   ```

3. Create the entry for the first operator mapping with a call to the `RTW.TflCOperationEntry` function.

   ```
   % Create table entry for matrix_sum_2x2_double
   op_entry = RTW.TflCOperationEntry;
   ```

4. Set operator entry parameters with a call to the `setTflCOperationEntryParameters` function. The code generator ignores saturation and rounding modes for floating-point nonscalar addition and subtraction. For code replacement entries for nonscalar addition and subtraction operations that do not involve fixed-point data, in the call to `setTflCOperationEntryParameters`, specify `'RTW_SATURATE_UNSPECIFIED'` for the `SaturationMode` property and `{'RTW_ROUND_UNSPECIFIED'}` for `RoundingModes`.

   ```
   setTflCOperationEntryParameters(op_entry, ...
       'Key',                    'RTW_OP_ADD', ...
       'Priority',               30, ...
       'SaturationMode',         'RTW_WRAP_ON_OVERFLOW', ...
       'ImplementationName',     'matrix_sum_2x2_double', ...
       'ImplementationHeaderFile', 'MatrixMath.h', ...
       'ImplementationSourceFile', 'MatrixMath.c', ...
       'ImplementationHeaderPath', LibPath, ...
       'ImplementationSourcePath', LibPath, ...
       'AdditionalIncludePaths',   {LibPath}, ...
       'GenCallback',            'RTW.copyFileToBuildDir', ...
       'SideEffects',            true);
   ```

5. Create conceptual arguments `y1`, `u1`, and `u2`. There are multiple ways to set up the conceptual arguments. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. To specify a matrix argument in the function call, use the argument

class `RTW.TflArgMatrix`. Specify the base type and the dimensions for which the argument is valid. The first table entry specifies `[2 2]` and the second table entry specifies `[3 3]`.

```
% Specify operands and result
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
                          'Name',         'y1', ...
                          'IOType',       'RTW_IO_OUTPUT', ...
                          'BaseType',     'double', ...
                          'DimRange',     [2 2]);
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix',...
                          'Name',         'u1', ...
                          'BaseType',     'double', ...
                          'DimRange',     [2 2]);
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix',...
                          'Name',         'u2', ...
                          'BaseType',     'double', ...
                          'DimRange',     [2 2]);
```

6 Create the implementation arguments. There are multiple ways to set up the implementation arguments. This example uses calls to the `getTflArgFromString` to create the arguments. The convenience methods `setReturn` and `addArgument` specify whether an argument is a return value or argument and adds the argument to the entry's array of implementation arguments.

```
arg = getTflArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.setReturn(arg);

arg = getTflArgFromString(hTable, 'u1', ['double' '*']);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u2', ['double' '*']);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'y1', ['double' '*']);
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.addArgument(arg);
```

7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

8 Create the entry for the second operator mapping.

```
% Create table entry for matrix_sum_3x3_double
op_entry = RTW.TflCOperationEntry;
setTflCOperationEntryParameters(op_entry, ...
    'Key',                      'RTW_OP_ADD', ...
    'Priority',                 30, ...
    'SaturationMode',           'RTW_WRAP_ON_OVERFLOW', ...
    'ImplementationName',       'matrix_sum_3x3_double', ...
```

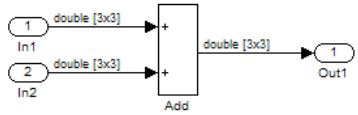
```
        'ImplementationHeaderFile', 'MatrixMath.h', ...
        'ImplementationSourceFile', 'MatrixMath.c', ...
        'ImplementationHeaderPath', LibPath, ...
        'ImplementationSourcePath', LibPath, ...
        'AdditionalIncludePaths',   {LibPath}, ...
        'GenCallback',              'RTW.copyFileToBuildDir', ...
        'SideEffects',              true);

    % Specify operands and result
    createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
                              'Name',         'y1', ...
                              'IOType',       'RTW_IO_OUTPUT', ...
                              'BaseType',     'double', ...
                              'DimRange',     [3 3]);
    createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix',...
                              'Name',         'u1', ...
                              'BaseType',     'double', ...
                              'DimRange',     [3 3]);
    createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix',...
                              'Name',         'u2', ...
                              'BaseType',     'double', ...
                              'DimRange',     [3 3]);

    % Specify replacement function signature
    arg = getTflArgFromString(hTable, 'y2', 'void');
    arg.IOType = 'RTW_IO_OUTPUT';
    op_entry.Implementation.setReturn(arg);
    arg = getTflArgFromString(hTable, 'u1', ['double' '*']);
    op_entry.Implementation.addArgument(arg);
    arg = getTflArgFromString(hTable, 'u2', ['double' '*']);
    op_entry.Implementation.addArgument(arg);
    arg = getTflArgFromString(hTable, 'y1', ['double' '*']);
    arg.IOType = 'RTW_IO_OUTPUT';
    op_entry.Implementation.addArgument(arg);

    addEntry(hTable, op_entry);
```

9 Save the table definition file. Use the name of the table definition function to name the file.

To test this example:

1 Register the code replacement mapping.

2 Create a model that includes an Add block.



3 Configure the model with the following settings:

- On the **Solver** pane, select a fixed-step, discrete solver with a fixed-step size such as `0.1`.
- On the **Code Generation** pane, select an ERT-based system target file.
- On the **Code Generation** > **Interface** pane, select the code replacement library that contains your addition operation entry.

4 In the Model Explorer, configure the **Signal Attributes** for the `In1` and `In2` source blocks. For each source block, set **Port dimensions** to `[3,3]`, and set **Data type** to `double`. Apply the changes. Save the model.

5 Generate code and a code generation report.

6 Review the code replacements. The code generator replaces the + operator with `matrix_sum_3x3_double` in the generated code.

7 Reconfigure **Port dimensions** for `In1` and `In2` to `[2 2]`, regenerate code. Observe that code containing the + operator is replaced with `matrix_sum_2x2_double`.

More About

- “Code You Can Replace From Simulink Models” on page 51-7
- “Define Code Replacement Mappings” on page 51-42
- “Matrix Multiplication Operation to MathWorks BLAS Code Replacement” on page 51-178
- “Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement” on page 51-186
- “Data Alignment for Code Replacement” on page 51-133
- “Remap Operator Output to Function Input” on page 51-192
- “Customize Match and Replacement Process” on page 51-153
- “Develop a Code Replacement Library” on page 51-27

Matrix Multiplication Operation to MathWorks BLAS Code Replacement

This example shows how to define code replacement mappings that replace nonscalar multiplication operations with Basic Linear Algebra Subroutine (BLAS) multiplication functions `xgemm` and `xgemv`. The example defines code replacement entries that map floating-point matrix/matrix and matrix/vector multiplication operations to MathWorks BLAS library multiplication functions `dgemm` and `dgemv`. The example defines the function mappings programmatically. Alternatively, you can use the Code Replacement Tool to define the same mappings.

BLAS libraries support matrix/matrix multiplication in the form of $C = a(op(A) * op(B)) + bC$. $op(X)$ means X, transposition of X, or Hermitian transposition of X. However, code replacement libraries support only the limited case of $C = op(A) * op(B)\ (a = 1.0, b = 0.0)$. Correspondingly, although BLAS libraries support matrix/vector multiplication in the form of $y = a(op(A) * x) + by$, code replacement libraries support only the limited case of $y = op(A) * x\ (a = 1.0, b = 0.0)$.

1. Create a table definition file that contains a function definition. For example:

   ```
   function hTable = crl_table_tmwblas_mmult_double
   ```

2. Within the function body, create the table by calling the function `RTW.TflTable`.

   ```
   hTable = RTW.TflTable;
   ```

3. Define the path for the BLAS function library. If your replacement functions are on the MATLAB search path or are in your working folder, you can skip this step.

   ```
   % Define library path for Windows or UNIX
   arch = computer('arch');
   if ~ispc
       LibPath = fullfile('$(MATLAB_ROOT)', 'bin', arch);
   else
       % Use Stateflow to get the compiler info
       compilerInfo = sf('Private','compilerman','get_compiler_info');
       compilerName = compilerInfo.compilerName;
       if strcmp(compilerName, 'msvc90') || ...
               strcmp(compilerName, 'msvc80') || ...
               strcmp(compilerName, 'msvc71') || ...
               strcmp(compilerName, 'msvc60'), ...
               compilerName = 'microsoft';
       end
       LibPath = fullfile('$(MATLAB_ROOT)', 'extern', 'lib', arch, compilerName);
   end
   ```

4 Create an entry for the first mapping with a call to the `RTW.TflBlasEntryGenerator` function.

```
% Create table entry for dgemm32
op_entry = RTW.TflBlasEntryGenerator;
```

5 Set operator entry parameters with a call to the `setTflCFunctionEntryParameters` function. The function call sets matrix multiplication operator entry properties. The code generator ignores saturation and rounding modes for floating-point nonscalar addition and subtraction. For code replacement entries for nonscalar addition and subtraction operations that do not involve fixed-point data, in the call to `setTflCFunctionEntryParameters`, specify `'RTW_SATURATE_UNSPECIFIED'` for the `SaturationMode` property and `{'RTW_ROUND_UNSPECIFIED'}` for `RoundingModes`.

```
if ispc
    libExt = 'lib';
elseif ismac
    libExt = 'dylib';
else
    libExt = 'so';
end
setTflCOperationEntryParameters(op_entry, ...
    'Key',                      'RTW_OP_MUL', ...
    'Priority',                 100, ...
    'ImplementationName',       'dgemm32', ...
    'ImplementationHeaderFile', 'blascompat32_crl.h', ...
    'ImplementationHeaderPath', fullfile('$(MATLAB_ROOT)','extern','include'), ...
    'AdditionalLinkObjs',       {['libmwblascompat32.' libExt]}, ...
    'AdditionalLinkObjsPaths',  {LibPath}, ...
    'SideEffects',              true);
```

6 Create conceptual arguments `y1`, `u1`, and `u2`. There are multiple ways to set up the conceptual arguments. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. To specify a matrix argument in the function call, use the argument class `RTW.TflArgMatrix` and specify the base type and the dimensions for which the argument is valid. This type of table entry supports a range of dimensions specified in the format `[Dim1Min Dim2Min ... DimNMin; Dim1Max Dim2Max ... DimNMax]`. For example, `[2 2; inf inf]` means a two-dimensional matrix of size 2x2 or larger. The conceptual output argument for the `dgemm32` entry for matrix/matrix multiplication replacement specifies dimensions `[2 2; inf inf]`, while the conceptual output argument for the `dgemv32` entry for matrix/vector multiplication replacement specifies dimensions `[2 1; inf 1]`.

```
% Specify operands and result
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
                          'Name',         'y1', ...
```

```
                            'IOType',       'RTW_IO_OUTPUT', ...
                            'BaseType',     'double', ...
                            'DimRange',     [2 2; inf inf]);
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
                            'Name',         'u1', ...
                            'BaseType',     'double', ...
                            'DimRange',     [2 2; inf inf]);
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
                            'Name',         'u2', ...
                            'BaseType',     'double', ...
                            'DimRange',     [1 1; inf inf]);
```

7 Create the implementation arguments. There are multiple ways to set up the implementation arguments. This example uses calls to the `getTflArgFromString` and `RTW.TflArgCharConstant` functions to create the arguments. The example code configures special implementation arguments that are required for `dgemm` and `dgemv` function replacements. The convenience methods `setReturn` and `addArgument` specify whether an argument is a return value or argument and adds the argument to the entry's array of implementation arguments.

```
% Using RTW.TflBlasEntryGenerator for xgemm requires the following
% implementation signature:
%
% void f(char* TRANSA, char* TRANSB, int* M, int* N, int* K,
%        type* ALPHA, type* u1, int* LDA, type* u2, int* LDB,
%        type* BETA, type* y, int* LDC)
%
% When a match occurs, the code generator computes the
% values for M, N, K, LDA, LDB, and LDC and inserts them into the
% generated code. TRANSA and TRANSB are set to 'N'.

% Specify replacement function signature

arg = getTflArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.setReturn(arg);

arg = RTW.TflArgCharConstant('TRANSA');
% Possible values for PassByType property are
%  RTW_PASSBY_AUTO, RTW_PASSBY_POINTER,
%  RTW_PASSBY_VOID_POINTER, RTW_PASSBY_BASE_POINTER
arg.PassByType = 'RTW_PASSBY_POINTER';
op_entry.Implementation.addArgument(arg);

arg = RTW.TflArgCharConstant('TRANSB');
arg.PassByType = 'RTW_PASSBY_POINTER';
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'M', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);
```

```
arg = getTflArgFromString(hTable, 'N', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'K', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'ALPHA', 'double', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u1', ['double' '*']);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'LDA', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u2', ['double' '*']);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'LDB', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'BETA', 'double', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'y1', ['double' '*']);
arg.IOType = 'RTW_IO_OUTPUT';
arg.PassByType = 'RTW_PASSBY_POINTER';
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'LDC', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);
```

8 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

9 Create the entry for the second mapping.

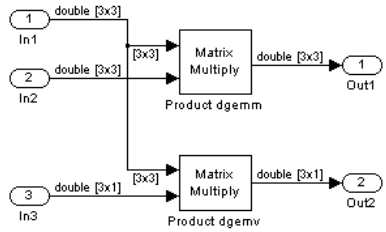
```
% Create table entry for dgemv32
op_entry = RTW.TflBlasEntryGenerator;
if ispc
    libExt = 'lib';
elseif ismac
    libExt = 'dylib';
else
    libExt = 'so';
end
setTflCOperationEntryParameters(op_entry, ...
    'Key',                    'RTW_OP_MUL', ...
    'Priority',               100, ...
    'ImplementationName',     'dgemv32', ...
    'ImplementationHeaderFile', 'blascompat32_crl.h', ...
    'ImplementationHeaderPath', fullfile('$(MATLAB_ROOT)','extern','include'), ...
    'AdditionalLinkObjs',     {['libmwblascompat32.' libExt]}, ...
    'AdditionalLinkObjsPaths',  {LibPath},...
    'SideEffects',            true);

% Specify operands and result
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
                          'Name',         'y1', ...
                          'IOType',       'RTW_IO_OUTPUT', ...
                          'BaseType',     'double', ...
                          'DimRange',     [2 1; inf 1]);
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
                          'Name',         'u1', ...
                          'BaseType',     'double', ...
                          'DimRange',     [2 2; inf inf]);
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix',...
                          'Name',         'u2', ...
                          'BaseType',     'double', ...
                          'DimRange',     [1 1; inf 1]);

% Using RTW.TflBlasEntryGenerator for xgemv requires the following
% implementation signature:
%
% void f(char* TRANS, int* M, int* N,
%        type* ALPHA, type* u1, int* LDA, type* u2, int* INCX,
%        type* BETA, type* y, int* INCY)
%
% Upon a match, the CRL entry will compute the
% values for M, N, LDA, INCX, and INCY, and insert them into the
% generated code. TRANS will be set to 'N'.

% Specify replacement function signature

arg = getTflArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.setReturn(arg);

arg = RTW.TflArgCharConstant('TRANS');
arg.PassByType = 'RTW_PASSBY_POINTER';
op_entry.Implementation.addArgument(arg);
```

```
arg = getTflArgFromString(hTable, 'M', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'N', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'ALPHA', 'double', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u1', ['double' '*']);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'LDA', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u2', ['double' '*']);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'INCX','integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'BETA', 'double', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'y1', ['double' '*']);
arg.IOType = 'RTW_IO_OUTPUT';
arg.PassByType = 'RTW_PASSBY_POINTER';
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'INCY', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

addEntry(hTable, op_entry);
```

10 Save the table definition file. Use the name of the table definition function to name the file.

To test this example:

1. Register the code replacement mapping.
2. Create a model that includes two Product blocks.



3. For each Product block, set the block parameter **Multiplication** to the value `Matrix(*)`.
4. Configure the model with the following settings:
   - On the **Solver** pane, select a fixed-step, discrete solver with a fixed-step size such as `0.1`.
   - On the **Code Generation** pane, select an ERT-based system target file.
   - On the **Code Generation** > **Interface** pane, select the code replacement library that contains your addition operation entry.
5. In the Model Explorer, configure the **Signal Attributes** for the `In1`, `In2`, and `In3` source blocks. For `In1` and `In2`, set **Port dimensions** to `[3 3]` and set the **Data type** to `double`. For `In3`, set **Port dimensions** to `[3 1]` and set the **Data type** to `double`.
6. Generate code and a code generation report.
7. Review the code replacements.

More About

- “Code You Can Replace From Simulink Models” on page 51-7
- “Define Code Replacement Mappings” on page 51-42
- “Small Matrix Operation to Processor Code Replacement” on page 51-174
- “Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement” on page 51-186
- “Data Alignment for Code Replacement” on page 51-133

- "Remap Operator Output to Function Input" on page 51-192
- "Customize Match and Replacement Process" on page 51-153
- "Develop a Code Replacement Library" on page 51-27

Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement

This example shows how to define code replacement mappings that replace nonscalar multiplication operations with ANSI/ISO C BLAS multiplication functions `xgemm` and `xgemv`. The example defines code replacement entries that map floating-point matrix/matrix and matrix/vector multiplication operations to ANSI/ISO C BLAS library multiplication functions `dgemm` and `dgemv`. The example defines the function mappings programmatically. Alternatively, you can use the Code Replacement Tool to define the same mappings.

BLAS libraries support matrix/matrix multiplication in the form of $C = a(op(A) * op(B)) + bC$. $op(X)$ means X, transposition of X, or Hermitian transposition of X. However, code replacement libraries support only the limited case of $C = op(A) * op(B)\ (a = 1.0, b = 0.0)$. Correspondingly, although BLAS libraries support matrix/vector multiplication in the form of $y = a(op(A) * x) + by$, code replacement libraries support only the limited case of $y = op(A) * x\ (a = 1.0, b = 0.0)$.

1. Create a table definition file that contains a function definition. For example:

   ```
   function hTable = crl_table_cblas_mmult_double
   ```

2. Within the function body, create the table by calling the function `RTW.TflTable`.

   ```
   hTable = RTW.TflTable;
   ```

3. Define the path for the CBLAS function library. For example:

   ```
   LibPath = fullfile(matlabroot, 'toolbox', 'rtw', 'rtwdemos', 'crl_demo');
   ```

4. Create an entry for the first mapping with a call to the `RTW.TflBlasEntryGenerator` function.

   ```
   % Create table entry for cblas_dgemm
   op_entry = RTW.TflCBlasEntryGenerator;
   ```

5. Set operator entry parameters with a call to the `setTflCOperationEntryParameters` function. The function call sets matrix multiplication operator entry properties. The code generator ignores saturation and rounding modes for floating-point nonscalar addition and subtraction.

   ```
   setTflCOperationEntryParameters(op_entry, ...
       'Key',                      'RTW_OP_MUL', ...
   ```

```
    'Priority',                100, ...
    'ImplementationName',      'cblas_dgemm', ...
    'ImplementationHeaderFile', 'cblas.h', ...
    'ImplementationHeaderPath', LibPath, ...
    'AdditionalIncludePaths',   {LibPath}, ...
    'GenCallback',             'RTW.copyFileToBuildDir', ...
    'SideEffects',             true);
```

6 Create conceptual arguments `y1`, `u1`, and `u2`. There are multiple ways to set up the conceptual arguments. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. To specify a matrix argument in the function call, use the argument class `RTW.TflArgMatrix` and specify the base type and the dimensions for which the argument is valid. This type of table entry supports a range of dimensions specified in the format `[Dim1Min Dim2Min ... DimNMin; Dim1Max Dim2Max ... DimNMax]`. For example, `[2 2; inf inf]` means a two-dimensional matrix of size 2x2 or larger. The conceptual output argument for the `dgemm32` entry for matrix/matrix multiplication replacement specifies dimensions `[2 2; inf inf]`. The conceptual output argument for the `dgemv32` entry for matrix/vector multiplication replacement specifies dimensions `[2 1; inf 1]`.

```
% Specify operands and result
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
                          'Name',          'y1', ...
                          'IOType',        'RTW_IO_OUTPUT', ...
                          'BaseType',      'double', ...
                          'DimRange',      [2 2; inf inf]);
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
                          'Name',          'u1', ...
                          'BaseType',      'double', ...
                          'DimRange',      [2 2; inf inf]);
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
                          'Name',          'u2', ...
                          'BaseType',      'double', ...
                          'DimRange',      [1 1; inf inf]);
```

7 Create the implementation arguments. There are multiple ways to set up the implementation arguments. This example uses calls to the `getTflArgFromString` function to create the arguments. The example code configures special implementation arguments that are required for `dgemm` and `dgemv` function replacements. The convenience methods `setReturn` and `addArgument` specify whether an argument is a return value or argument and adds the argument to the entry's array of implementation arguments.

```
% Using RTW.TflCBlasEntryGenerator for xgemm requires the following
% implementation signature:
%
% void f(enum ORDER, enum TRANSA, enum TRANSB, int M, int N, int K,
%        type ALPHA, type* u1, int LDA, type* u2, int LDB,
```

```
%        type BETA, type* y, int LDC)
%
% Since CRLs do not have the ability to specify enums, you must
% use integer.  (This will cause problems with C++ code generation,
% so for C++, use a wrapper function to cast each int to the
% corresponding enumeration type.)
%
% When a match occurs, the code generator computes the
% values for M, N, K, LDA, LDB, and LDC and insert them into the
% generated code.

% Specify replacement function signature

arg = getTflArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.setReturn(arg);

arg = getTflArgFromString(hTable, 'ORDER', 'integer', 102);
% arg.Type.ReadOnly=true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'TRANSA', 'integer', 111);
% arg.Type.ReadOnly=true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'TRANSB', 'integer', 111);
% arg.Type.ReadOnly=true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'M', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'N', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'K', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'ALPHA', 'double', 1);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u1', ['double' '*']);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'LDA', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u2', ['double' '*']);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'LDB', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'BETA', 'double', 0);
op_entry.Implementation.addArgument(arg);
```

```
arg = getTflArgFromString(hTable, 'y1', ['double' '*']);
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'LDC', 'integer', 0);
op_entry.Implementation.addArgument(arg);
```

8 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

9 Create the entry for the second mapping.

```
% Create table entry for cblas_dgemv
op_entry = RTW.TflCBlasEntryGenerator;
setTflCOperationEntryParameters(op_entry, ...
    'Key',                      'RTW_OP_MUL', ...
    'Priority',                 100, ...
    'ImplementationName',       'cblas_dgemv', ...
    'ImplementationHeaderFile', 'cblas.h', ...
    'ImplementationHeaderPath', LibPath, ...
    'AdditionalIncludePaths',   {LibPath}, ...
    'GenCallback',              'RTW.copyFileToBuildDir', ...
    'SideEffects',              true);

% Specify operands and result
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
                          'Name',         'y1', ...
                          'IOType',       'RTW_IO_OUTPUT', ...
                          'BaseType',     'double', ...
                          'DimRange',     [2 1; inf 1]);
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
                          'Name',         'u1', ...
                          'BaseType',     'double', ...
                          'DimRange',     [2 2; inf inf]);
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix',...
                          'Name',         'u2', ...
                          'BaseType',     'double', ...
                          'DimRange',     [1 1; inf 1]);

% Using RTW.TflCBlasEntryGenerator for xgemv requires the following
% implementation signature:
%
% void f(enum ORDER, enum TRANSA, int M, int N,
%        type ALPHA, type* u1, int LDA, type* u2, int INCX,
%        type BETA, type* y, int INCY)
%
% Since CRLs do not have the ability to specify enums, you must
% use integer.  (This will cause problems with C++ code generation,
% so for C++, use a wrapper function to cast each int to the
% corresponding enumeration type.)
%
% Upon a match, the CRL entry will compute the
% values for M, N, LDA, INCX, and INCY and insert them into the
% generated code.
```

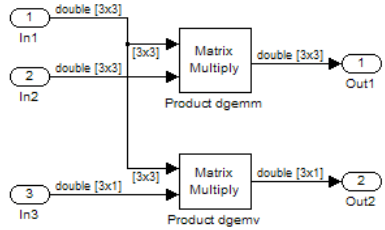
```
% Specify replacement function signature

arg = getTflArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.setReturn(arg);

arg = getTflArgFromString(hTable, 'ORDER', 'integer', 102);
% arg.Type.ReadOnly=true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'TRANSA', 'integer', 111);
% arg.Type.ReadOnly=true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'M','integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'N', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'ALPHA', 'double', 1);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u1', ['double' '*']);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'LDA', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u2', ['double' '*']);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'INCX', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'BETA', 'double', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'y1', ['double' '*']);
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'INCY', 'integer', 0);
op_entry.Implementation.addArgument(arg);

addEntry(hTable, op_entry);
```

10 Save the table definition file. Use the name of the table definition function to name the file.

To test this example:

1 Register the code replacement mapping.

2 Create a model that includes two Product blocks.



3 Configure the model with the following settings:

- On the **Solver** pane, select a fixed-step, discrete solver with a fixed-step size such as `0.1`.
- On the **Code Generation** pane, select an ERT-based system target file.
- On the **Code Generation** > **Interface** pane, select the code replacement library that contains your addition operation entry.

4 For each Product block, set the block parameter **Multiplication** to the value `Matrix(*)`.

5 In the Model Explorer, configure the **Signal Attributes** for the `In1`, `In2`, and `In3` source blocks. For `In1` and `In2`, set **Port dimensions** to `[3 3]`. Set the **Data type** to `double`. For `In3`, set **Port dimensions** to `[3 1]`. Set the **Data type** to `double`.

6 Generate code and a code generation report.

7 Review the code replacements.

More About

- "Code You Can Replace From Simulink Models" on page 51-7
- "Define Code Replacement Mappings" on page 51-42
- "Small Matrix Operation to Processor Code Replacement" on page 51-174
- "Matrix Multiplication Operation to MathWorks BLAS Code Replacement" on page 51-178
- "Data Alignment for Code Replacement" on page 51-133
- "Remap Operator Output to Function Input" on page 51-192
- "Customize Match and Replacement Process" on page 51-153
- "Develop a Code Replacement Library" on page 51-27

Remap Operator Output to Function Input

If your generated code must meet a specific coding pattern or you want more flexibility, for example, to further improve performance, you can remap operator outputs to input positions in an implementation function argument list.

Note: Remapping outputs to implementation function inputs is supported only for operator replacement.

For example, for a sum operation, the code generator produces code similar to:

```
add8_Y.Out1 = u8_add_u8_u8(add8_U.In1, add8_U.In2);
```

If you remap the output to the first input, the code generator produces code similar to:

```
u8_add_u8_u8(&add8_Y.Out1;, add8_U.In1, add8_U.In2);
```

The following table definition file for a sum operation remaps operator output `y1` as the first function input argument.

1. Create a table definition file that contains a function definition. For example:

   ```
   function hTable = crl_table_add_uint8
   ```

2. Within the function body, create the table by calling the function `RTW.TflTable`.

   ```
   hTable = RTW.TflTable;
   ```

3. Create an entry for the operator mapping with a call to the `RTW.TflCOperationEntry` function.

   ```
   % Create operation entry
   op_entry = RTW.TflCOperationEntry;
   ```

4. Set operator entry parameters with a call to the `setTflCOperationEntryParameters` function. In the function call, set the property `SideEffects` to `true`.

   ```
   setTflCOperationEntryParameters(op_entry, ...
                       'Key',                      'RTW_OP_ADD', ...
                       'Priority',                 90, ...
                       'ImplementationName',       'u8_add_u8_u8', ...
                       'ImplementationHeaderFile', 'u8_add_u8_u8.h', ...
                       'ImplementationSourceFile', 'u8_add_u8_u8.c', ...
                       'SideEffects',              true );
   ```

5 Create conceptual arguments `y1`, `u1`, and `u2`. There are multiple ways to set up the conceptual arguments. This example uses calls to the `getTflArgFromString` and `addConceptualArg` functions to create and add the arguments.

```
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 );
```

6 Create the implementation arguments. There are multiple ways to set up the implementation arguments. This example uses calls to the `getTflArgFromString` function to create the arguments. When defining the implementation function return argument, create a new `void` output argument, for example, `y2`. When defining the implementation function argument for the conceptual output argument (`y1`), set the operator output argument as an additional input argument. Mark its `IOType` as output. Make its type a pointer type. The convenience methods `setReturn` and `addArgument` specify whether an argument is a return value or argument and adds the argument to the entry's array of implementation arguments.

```
% Create new void output y2
arg = getTflArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.setReturn(arg);

% Set y1 as first input arg, mark IOType as output, and use pointer type
arg=getTflArgFromString(hTable, 'y1', 'uint8*');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.addArgument(arg);

arg=getTflArgFromString(hTable, 'u1', 'uint8');
op_entry.Implementation.addArgument(arg);

arg=getTflArgFromString(hTable, 'u2', 'uint8');
op_entry.Implementation.addArgument(arg);
```

7 Add the entry to a code replacement table with a call to the `addEntry` function.

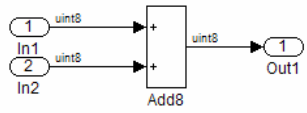
```
addEntry(hTable, op_entry);
```

8 Save the table definition file. Use the name of the table definition function to name the file.

To test this example:

1 Register the code replacement mapping.

2 Create a model that includes an Add block.



3 Configure the model with the following settings:

- On the **Solver** pane, select a fixed-step solver.
- On the **Code Generation** pane, select an ERT-based system target file.
- On the **Code Generation** > **Interface** pane, select the code replacement library that contains your addition operation entry.
- On the **All Parameters** tab, set the **Optimize global data access** parameter to `Use global to hold temporary results` to reduce data copies in the generated code.

4 Generate code and a code generation report.

5 Review the code replacements.

More About

- "Code You Can Replace From Simulink Models" on page 51-7
- "Define Code Replacement Mappings" on page 51-42
- "Develop a Code Replacement Library" on page 51-27

Fixed-Point Operator Code Replacement

If you have a Fixed-Point Designer license, you can define fixed-point operator code replacement entries to match:

- A binary-point-only scaling combination on the operator inputs and output.
- A slope bias scaling combination on the operator inputs and output.
- Relative scaling or net slope between multiplication or division operator inputs and output. Use one of these methods to map a range of slope and bias values to a replacement function for multiplication or division.
- Equal slope and zero net bias across addition or subtraction operator inputs and output. Use this method to disregard specific slope and bias values and map relative slope and bias values to a replacement function for addition or subtraction.

Common Ways to Match Fixed-Point Operator Entries

The following table maps common ways to match fixed-point operator code replacement entries with the associated fixed-point parameters that you specify in a code replacement table definition file.

Match	Create entry	Minimally specify parameters
A specific binary-point-only scaling combination on the operator inputs and output.	`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.	`RTW.TflCOperationEntry`	`createAndAddConceptualArg` function:

Match	Create entry	Minimally specify parameters
		• `CheckSlope`: Specify the value `true`. • `CheckBias`: Specify the value `true`. • `DataTypeMode` (or `DataType/Scaling` equivalent): Specify fixed-point [slope bias] scaling. • `Slope` (or `SlopeAdjustmentFactor/FixedExponent` equivalent): Specify a slope value (for example, `15`). • `Bias`: Specify a bias value (for example, `2`).
Net slope between operator inputs and output (multiplication and division).	`RTW.TflCOperationEntryGenerator_NetSlope`	`setTflCOperationEntryParameters` function: • `NetSlopeAdjustmentFactor`: Specify the slope adjustment factor (`F`) part of the net slope, $F2^E$ (for example, `1.0`). • `NetFixedExponent`: Specify the fixed exponent (`E`) part of the net slope, $F2^E$ (for example, `-3.0`). `createAndAddConceptualArg` function: • `CheckSlope`: Specify the value `false`. • `CheckBias`: Specify the value `false`. • `DataType`: Specify the value `'Fixed'`.

Match	Create entry	Minimally specify parameters
Relative scaling between operator inputs and output (multiplication and division).	`RTW.TflCOperationEntryGenerator`	`setTflCOperationEntryParameters` function: • `RelativeScalingFactorF`: Specify the slope adjustment factor (`F`) part of the relative scaling factor, $F2^E$ (for example, `1.0`). • `RelativeScalingFactorE`: Specify the fixed exponent (`E`) part of the relative scaling factor, $F2^E$ (for example, `-3.0`). `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).	`RTW.TflCOperationEntryGenerator`	`setTflCOperationEntryParameters` function: • `SlopesMustBeTheSame`: Specify the value `true`. • `MustHaveZeroNetBias`: Specify the value `true`. `createAndAddConceptualArg` function: • `CheckSlope`: Specify the value `false`. • `CheckBias`: Specify the value `false`.

Fixed-Point Numbers and Arithmetic

Fixed-point numbers use integers and integer arithmetic to represent real numbers and arithmetic with the following encoding scheme:

$$V = \tilde{V} = SQ + B$$

- 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:

$$\left(S_O Q_O + B_O\right) = \left(S_1 Q_1 + B_1\right) < op > (S_2 Q_2 + B_2)$$

The objective of 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. The following sections provide additional programming information for each supported operator.

Addition

The operation $V_0 = V_1 + V_2$ implies that

$$Q_0 = \left(\frac{S_1}{S_0}\right) Q_1 + \left(\frac{S_2}{S_0}\right) Q_2 + \left(\frac{B_1 + B_2 - B_0}{S_0}\right)$$

If an addition replacement function is defined such that the scaling on the operands and sum are equal and the net bias

$$\left(\frac{B_1 + B_2 - B_0}{S_0} \right)$$

is zero (for example, a function `s8_add_s8_s8` that adds two signed 8-bit values and produces a signed 8-bit result), then the 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 $V_0 = V_1 - V_2$ implies that

$$Q_0 = \left(\frac{S_1}{S_0} \right) Q_1 - \left(\frac{S_2}{S_0} \right) Q_2 + \left(\frac{B_1 - B_2 - B_0}{S_0} \right)$$

If a subtraction replacement function is defined such that the scaling on the operands and difference are equal and the net bias

$$\left(\frac{B_1 - B_2 - B_0}{S_0} \right)$$

is zero (for example, a function `s8_sub_s8_s8` that subtracts two signed 8-bit values and produces a signed 8-bit result), then the 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

There are different ways to specify multiplication replacements. The most direct way is to specify an exact match of the input and output types. This is feasible if a model contains only a few (known) slope and bias combinations. Use the `TflCOperationEntry` class

and specify the exact values of slope and bias on each argument. For scenarios where there are numerous slope/bias combinations, it is not feasible to specify each value with a different entry. Use a net slope entry or create a custom entry.

The operation $V_0 = V_1 * V_2$ implies, for binary-point-only scaling, that

$$S_0Q_0 = (S_1Q_1)(S_2Q_2)$$
$$Q_0 = \left(\frac{S_1S_2}{S_0}\right)Q_1Q_2$$
$$Q_0 = S_nQ_1Q_2$$

where S_n is the net slope.

It is common to replace all multiplication operations that have a net slope of 1.0 with a function that performs C-style multiplication. For example, to replace all signed 8-bit multiplications that have a net scaling of 1.0 with the `s8_mul_s8_u8_` replacement function, the operator entry must define a net slope factor, $F2^E$. You specify the values for F and E using operator entry parameters `NetSlopeAdjustmentFactor` and `NetFixedExponent`. (For parameter descriptions, see the reference page for the function `setTflCOperationEntryParameters`.) For the `s8_mul_s8_u8` function, set `NetSlopeAdjustmentFactor` to 1 and `NetFixedExponent` to 0.0.

Note: When an operator entry specifies `NetSlopeAdjustmentFactor` and `NetFixedExponent`, matching entries must have arguments with zero bias.

Division

There are different ways to specify division replacements. The most direct way is to specify an exact match of the input and output types. This is feasible if a model contains only a few (known) slope and bias combinations. For this, use the `TflCOperationEntry` class and specify the exact values of slope and bias on each argument. For scenarios where there are numerous slope/bias combinations, it is not feasible to specify each value with a different entry. For this, use a net slope entry or create a custom entry (see "Customize Match and Replacement Process" on page 51-153).

The operation $V_0 = (V_1 / V_2)$ implies, for binary-point-only scaling, that

$$S_0 Q_0 = \left(\frac{S_1 Q_1}{S_2 Q_2} \right)$$

$$Q_0 = S_n \left(\frac{Q_1}{Q_2} \right)$$

where S_n is the net slope.

It is common to replace all division operations that have a net slope of 1.0 with a function that performs C-style division. For example, to replace all signed 8-bit divisions that have a net scaling of 1.0 with the `s8_mul_s8_u8_` replacement function, the operator entry must define a net slope factor, $F2^E$. You specify the values for F and E using operator entry parameters `NetSlopeAdjustmentFactor` and `NetFixedExponent`. (For parameter descriptions, see the reference page for the function `setTflCOperationEntryParameters`.) For the `s16_netslopeOp5_div_s16_s16` function, you would set `NetSlopeAdjustmentFactor` to 1 and `NetFixedExponent` to 0.0.

Note: When an operator entry specifies `NetSlopeAdjustmentFactor` and `NetFixedExponent`, matching entries must have arguments with zero bias.

Data Type Conversion (Cast)

The data type conversion operation $V_0 = V_1$ implies, for binary-point-only scaling, that

$$Q_0 = \left(\frac{S_1}{S_0} \right) Q_1$$

$$Q_0 = S_n Q_1$$

where S_n is the net slope.

Shift

The shift left or shift right operation $V_0 = (V_1 / 2^n)$ implies, for binary-point-only scaling, that

$$S_0 Q_0 = \left(\frac{S_1 Q_1}{2^n} \right)$$

$$Q_0 = \left(\frac{S_1}{S_0} \right) + \left(\frac{Q_1}{2^n} \right)$$

$$Q_0 = S_n \left(\frac{Q_1}{2^n} \right)$$

where S_n is the net slope.

More About

- "Code You Can Replace From Simulink Models" on page 51-7
- "Define Code Replacement Mappings" on page 51-42
- "Binary-Point-Only Scaling Code Replacement" on page 51-203
- "Slope Bias Scaling Code Replacement" on page 51-207
- "Net Slope Scaling Code Replacement" on page 51-211
- "Equal Slope and Zero Net Bias Code Replacement" on page 51-218
- "Data Type Conversions (Casts) and Operator Code Replacement" on page 51-222
- "Shift Left Operations and Code Replacement" on page 51-226
- "Data Alignment for Code Replacement" on page 51-133
- "Remap Operator Output to Function Input" on page 51-192
- "Customize Match and Replacement Process" on page 51-153
- "Develop a Code Replacement Library" on page 51-27

Binary-Point-Only Scaling Code Replacement

You can define code replacement entries for operations on fixed-point data types such that they match a binary-point-only scaling combination on 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 creates a code replacement entry for multiplication of fixed-point data types. You specify arguments using binary-point-only scaling. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

1. Create a table definition file that contains a function definition. For example:

   ```
   function hTable = crl_table_fixed_binptscale
   ```

2. Within the function body, create the table by calling the function `RTW.TflTable`.

   ```
   hTable = RTW.TflTable;
   ```

3. Create the entry for the operator mapping with a call to the `RTW.TflCOperationEntry` function.

   ```
   op_entry = RTW.TflCOperationEntry;
   ```

4. Set operator entry parameters with a call to the `setTflCOperationEntryParameters` function. The parameters specify the type of operation as multiplication, the saturation mode as saturate on integer overflow, rounding modes as unspecified, and the name of the replacement function as `s32_mul_s16_s16_binarypoint`.

   ```
   setTflCOperationEntryParameters(op_entry, ...
       'Key',                    'RTW_OP_MUL', ...
       'Priority',               90, ...
       'SaturationMode',         'RTW_SATURATE_ON_OVERFLOW', ...
       'RoundingModes',          {'RTW_ROUND_UNSPECIFIED'}, ...
       'ImplementationName',     's32_mul_s16_s16_binarypoint', ...
       'ImplementationHeaderFile', 's32_mul_s16_s16_binarypoint.h', ...
       'ImplementationSourceFile', 's32_mul_s16_s16_binarypoint.c');
   ```

5. Create conceptual arguments `y1`, `u1`, and `u2`. There are multiple ways to set up the conceptual arguments. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. 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. The input arguments are 16 bits, signed, with fraction lengths of 15 and 13.

```
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);
```

6 Create the implementation arguments. There are multiple ways to set up the implementation arguments. This example uses calls to the `createAndSetCImplementationReturn` and `createAndAddImplementationArg` functions to create and add implementation arguments to the 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`). The input arguments are 16 bits and signed (`int16`).

```
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);
```

7 Add the entry to a code replacement table with a call to the `addEntry` function.

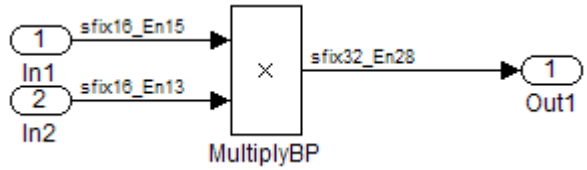
```
addEntry(hTable, op_entry);
```

8 Save the table definition file. Use the name of the table definition function to name the file.

To test this example:

1 Register the code replacement mapping.

2 Create a model.



3 For this model:

- Set the Inport 1 **Data type** to `fixdt(1,16,15)`.
- Set the Inport 2 **Data type** to `fixdt(1,16,13)`.
- In the Product block:
  - Set **Output data type** to `fixdt(1,32,28)`.
  - Select the option **Saturate on integer overflow**.

4 Configure the model with the following settings:

- On the **Solver** pane, select a fixed-step, discrete solver.
- On the **Code Generation** pane, select an ERT-based system target file.
- On the **Code Generation** > **Interface** pane, select the code replacement library that contains your addition operation entry.

5 Generate code and a code generation report.

6 Review the code replacements.

More About

- “Code You Can Replace From Simulink Models” on page 51-7
- “Define Code Replacement Mappings” on page 51-42
- “Fixed-Point Operator Code Replacement” on page 51-195
- “Data Type Conversions (Casts) and Operator Code Replacement” on page 51-222
- “Shift Left Operations and Code Replacement” on page 51-226
- “Data Alignment for Code Replacement” on page 51-133
- “Remap Operator Output to Function Input” on page 51-192
- “Customize Match and Replacement Process” on page 51-153
- “Develop a Code Replacement Library” on page 51-27

Slope Bias Scaling Code Replacement

You can define code replacement for operations on fixed-point data types as matching a slope bias scaling combination on the operator inputs and output. The slope bias scaling entries can map the specified slope bias combination to a replacement function for addition, subtraction, multiplication, or division.

This example creates a code replacement entry for division of fixed-point data types. You specify arguments using slope bias scaling. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

1. Create a table definition file that contains a function definition. For example:

   ```
   function hTable = crl_table_fixed_s16divslopebias
   ```

2. Within the function body, create the table by calling the function `RTW.TflTable`.

   ```
   hTable = RTW.TflTable;
   ```

3. Create the entry for the operator mapping with a call to the `RTW.TflCOperationEntry` function.

   ```
   op_entry = RTW.TflCOperationEntry;
   ```

4. Set operator entry parameters with a call to the `setTflCOperationEntryParameters` function. The parameters specify the type of operation as division, the saturation mode as saturate on integer overflow, rounding modes as round to ceiling, and the name of the replacement function as `s16_div_s16_s16_slopebias`.

   ```
   setTflCOperationEntryParameters(op_entry, ...
      'Key',                    'RTW_OP_DIV', ...
      'Priority',               90, ...
      'SaturationMode',         'RTW_SATURATE_ON_OVERFLOW', ...
      'RoundingModes',          {'RTW_ROUND_CEILING'}, ...
      'ImplementationName',     's16_div_s16_s16_slopebias', ...
      'ImplementationHeaderFile', 's16_div_s16_s16_slopebias.h', ...
      'ImplementationSourceFile', 's16_div_s16_s16_slopebias.c');
   ```

5. Create conceptual arguments `y1`, `u1`, and `u2`. There are multiple ways to set up the conceptual arguments. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. 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.

```
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);
```

6 Create the implementation arguments. There are multiple ways to set up the implementation arguments. This example uses calls to the `createAndSetCImplementationReturn` and `createAndAddImplementationArg` functions to create and add implementation arguments to the 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`).

```
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);
```

7 Add the entry to a code replacement table with a call to the `addEntry` function.

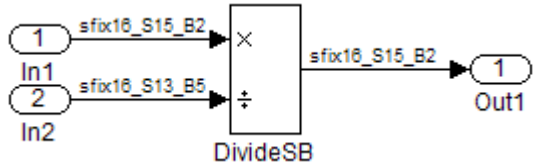
```
addEntry(hTable, op_entry);
```

8 Save the table definition file. Use the name of the table definition function to name the file.

To test this example:

1 Register the code replacement mapping.

2 Create a model.



3 For this model:

- Set the Inport 1 **Data type** to `fixdt(1,16,15,2)`.
- Set the Inport 2 **Data type** to `fixdt(1,16,13,5)`.
- In the Divide block:
  - Set **Output data type** to `Inherit: Inherit via back propagation`.
  - Set **Integer rounding mode** to `Ceiling`.
  - Select the option **Saturate on integer overflow**.

4 Configure the model with the following settings:

- On the **Solver** pane, select a fixed-step, discrete solver.
- On the **Code Generation** pane, select an ERT-based system target file.
- On the **Code Generation** > **Interface** pane, select the code replacement library that contains your addition operation entry.

5 Generate code and a code generation report.

6 Review the code replacements.

More About

- "Code You Can Replace From Simulink Models" on page 51-7
- "Define Code Replacement Mappings" on page 51-42
- "Fixed-Point Operator Code Replacement" on page 51-195
- "Data Type Conversions (Casts) and Operator Code Replacement" on page 51-222
- "Shift Left Operations and Code Replacement" on page 51-226
- "Data Alignment for Code Replacement" on page 51-133
- "Remap Operator Output to Function Input" on page 51-192
- "Customize Match and Replacement Process" on page 51-153
- "Develop a Code Replacement Library" on page 51-27

Net Slope Scaling Code Replacement

Multiplication and Division with Saturation

You can define code replacement entries for operations on fixed-point data types as matching net slope between operator inputs and output. The net slope entries can map a range of slope and bias values to a replacement function for multiplication or division.

This example creates a code replacement entry for division of fixed-point data types, using wrap on overflow saturation mode and a net slope. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

1. Create a table definition file that contains a function definition. For example:

   ```
   function hTable = crl_table_fixed_netslopesaturate
   ```

2. Within the function body, create the table by calling the function `RTW.TflTable`.

   ```
   hTable = RTW.TflTable;
   ```

3. Create the entry for the operator mapping with a call to the `RTW.TflCOperationEntryGenerator_Netslope` function, which provides access to the fixed-point parameters `NetSlopeAdjustmentFactor` and `NetFixedExponent`.

   ```
   wv = [16,32];
   for iy = 1:2
     for inum = 1:2
       for iden = 1:2
         hTable = getDivOpEntry(hTable, ...
            fixdt(1,wv(iy)),fixdt(1,wv(inum)),fixdt(1,wv(iden)));
       end
     end
   end


   %------------------------------------------------------------
   function hTable = getDivOpEntry(hTable,dty,dtnum,dtden)
   %------------------------------------------------------------
   % Create an entry for division of fixed-point data types where
   % arguments are specified using Slope and Bias scaling
   % Saturation on, Rounding unspecified

   funcStr = sprintf('user_div_%s_%s_%s',...
           typeStrFunc(dty),...
           typeStrFunc(dtnum),...
           typeStrFunc(dtden));
   ```

```
op_entry = RTW.TflCOperationEntryGenerator_NetSlope;
```

4 Set operator entry parameters with a call to the `setTflCOperationEntryParameters` function. The parameters specify the type of operation as division, the saturation mode as wrap on overflow, rounding modes as unspecified, and the name of the replacement function as `user_div_*`. `NetSlopeAdjustmentFactor` and `NetFixedExponent` specify the `F` and `E` parts of the net slope $F2^E$.

```
setTflCOperationEntryParameters(op_entry, ...
   'Key',                      'RTW_OP_DIV', ...
   'Priority',                 90, ...
   'SaturationMode',           'RTW_WRAP_ON_OVERFLOW',...
   'RoundingModes',            {'RTW_ROUND_UNSPECIFIED'},...
   'NetSlopeAdjustmentFactor', 1.0, ...
   'NetFixedExponent',         0.0, ...
   'ImplementationName',       funcStr, ...
   'ImplementationHeaderFile', [funcStr,'.h'], ...
   'ImplementationSourceFile', [funcStr,'.c']);
```

5 Create conceptual arguments `y1`, `u1`, and `u2`. There are multiple ways to set up the conceptual arguments. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. Specify each argument as fixed-point and signed. Also, for each argument, specify that code replacement request processing does *not* check for an exact match to the call-site slope and bias values.

```
createAndAddConceptualArg(op_entry, ...
   'RTW.TflArgNumeric', ...
   'Name',           'y1',...
   'IOType',         'RTW_IO_OUTPUT',...
   'CheckSlope',     false,...
   'CheckBias',      false,...
   'DataTypeMode',   'Fixed-point: slope and bias scaling',...
   'IsSigned',       dty.Signed,...
   'WordLength',     dty.WordLength,...
   'Bias',           0);

createAndAddConceptualArg(op_entry, ...
   'RTW.TflArgNumeric',...
   'Name',           'u1', ...
   'IOType',         'RTW_IO_INPUT',...
   'CheckSlope',     false,...
   'CheckBias',      false,...
   'DataTypeMode',   'Fixed-point: slope and bias scaling',...
   'IsSigned',       dtnum.Signed,...
   'WordLength',     dtnum.WordLength,...
   'Bias',           0);

createAndAddConceptualArg(op_entry, ...
   'RTW.TflArgNumeric', ...
   'Name',           'u2', ...
```

```
    'IOType',          'RTW_IO_INPUT',...
    'CheckSlope',      false,...
    'CheckBias',       false,...
    'DataTypeMode',    'Fixed-point: slope and bias scaling',...
    'IsSigned',        dtden.Signed,...
    'WordLength',      dtden.WordLength,...
    'Bias',            0);
```

6 Create the implementation arguments. There are multiple ways to set up the implementation arguments. This example uses calls to the `getTflArgFromString` function to create the arguments. Implementation arguments must describe fundamental numeric data types (not fixed-point data types). The convenience methods `setReturn` and `addArgument` specify whether an argument is a return value or argument. These methods add the argument to the entry array of implementation arguments.

```
arg = getTflArgFromString(hTable, 'y1', typeStrBase(dty));
op_entry.Implementation.setReturn(arg);

arg = getTflArgFromString(hTable, 'u1', typeStrBase(dtnum));
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u2',typeStrBase(dtden));
op_entry.Implementation.addArgument(arg);
```

7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

8 Define functions that determine the data type word length.

```
%------------------------------------------------------------
function str = typeStrFunc(dt)
%------------------------------------------------------------

if dt.Signed
    sstr = 's';
else
    sstr = 'u';
end
str = sprintf('%s%d',sstr,dt.WordLength);

%------------------------------------------------------------
function str = typeStrBase(dt)
%------------------------------------------------------------

if dt.Signed
    sstr = ;
else
    sstr = 'u';
end
str = sprintf('%sint%d',sstr,dt.WordLength);
```

9 Save the table definition file. Use the name of the table definition function to name the file.

Multiplication and Division with Rounding Mode and Additional Implementation Arguments

You can define code replacement entries for multiplication and division operations on fixed-point data types such that they match the net slope between operator inputs and output. The net slope entries can map a range of slope and bias values to a replacement function for multiplication or division.

This example creates a code replacement entry for division of fixed-point data types, using the ceiling rounding mode and a net slope scaling factor. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

1 Create a table definition file that contains a function definition. For example:

```
function hTable = crl_table_fixed_netsloperound
```

2 Within the function body, create the table by calling the function `RTW.TflTable`.

```
hTable = RTW.TflTable;
```

3 Create the entry for the operator mapping with a call to the `RTW.TflCOperationEntryGenerator_Netslope` function, which provides access to the fixed-point parameters `NetSlopeAdjustmentFactor` and `NetFixedExponent`.

```
op_entry = RTW.TflCOperationEntryGenerator_NetSlope;
```

4 Set operator entry parameters with a call to the `setTflCOperationEntryParameters` function. The parameters specify the type of operation as division, the saturation mode as saturation off, rounding modes as round to ceiling, and the name of the replacement function as `s16_div_s16_s16`. `NetSlopeAdjustmentFactor` and `NetFixedExponent` specify the `F` and `E` parts of the relative scaling factor $F2^E$.

```
setTflCOperationEntryParameters(op_entry, ...
   'Key',                      'RTW_OP_DIV', ...
   'Priority',                 90, ...
   'SaturationMode',           'RTW_WRAP_ON_OVERFLOW', ...
   'RoundingModes',            {'RTW_ROUND_CEILING'}, ...
   'NetSlopeAdjustmentFactor',   1.0, ...
   'NetFixedExponent',           0.0, ...
```

```
    'ImplementationName',       's16_div_s16_s16', ...
    'ImplementationHeaderFile', 's16_div_s16_s16.h', ...
    'ImplementationSourceFile', 's16_div_s16_s16.c');
```

5 Create conceptual arguments `y1`, `u1`, and `u2`. There are multiple ways to set up the conceptual arguments. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. Specify each argument as fixed-point, 16 bits, and signed. Also, for each argument, specify that code replacement request processing does *not* check for an exact match to the call-site slope and bias values.

```
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);
```

6 Create the implementation arguments. There are multiple ways to set up the implementation arguments. This example uses calls to the `createAndSetCImplementationReturn` and `createAndAddImplementationArg` functions to create and add implementation arguments to the 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`).

```
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);
```

7 Add the entry to a code replacement table with a call to the `addEntry` function.

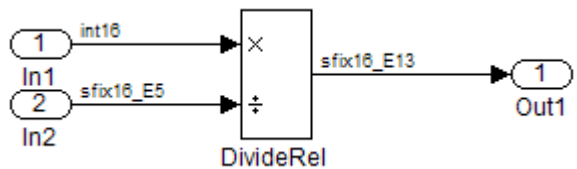
```
addEntry(hTable, op_entry);
```

8 Save the table definition file. Use the name of the table definition function to name the file.

To test this example:

1 Register the code replacement mapping.

2 Create a model.



3 For this model:

- Set the Inport 1 **Data type** to `int16`.
- Set the Inport 2 **Data type** to `fixdt(1,16,-5)`.
- In the Divide block:
  - Set **Output data type** to `fixdt(1,16,-13)`.
  - Set **Integer rounding mode** to `Ceiling`.

4 Configure the model with the following settings:

- On the **Solver** pane, select a fixed-step, discrete solver.

- On the **Code Generation** pane, select an ERT-based system target file.
- On the **Code Generation > Interface** pane, select the code replacement library that contains your addition operation entry.

5 Generate code and a code generation report.

6 Review the code replacements.

More About

- "Code You Can Replace From Simulink Models" on page 51-7
- "Define Code Replacement Mappings" on page 51-42
- "Fixed-Point Operator Code Replacement" on page 51-195
- "Data Type Conversions (Casts) and Operator Code Replacement" on page 51-222
- "Shift Left Operations and Code Replacement" on page 51-226
- "Data Alignment for Code Replacement" on page 51-133
- "Remap Operator Output to Function Input" on page 51-192
- "Customize Match and Replacement Process" on page 51-153
- "Develop a Code Replacement Library" on page 51-27

Equal Slope and Zero Net Bias Code Replacement

You can define code replacement entries for addition or subtraction of fixed-point data types such that they match relative slope and bias values (equal slope and zero net bias) across operator inputs and output. These entries allow you to disregard slope and bias values. Map relative slope and bias values to a replacement function for addition or subtraction.

This example creates a code replacement entry for addition of fixed-point data types. Slopes must be equal and net bias must be zero across the operator inputs and output. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

1. Create a table definition file that contains a function definition. For example:

   ```
   function hTable = crl_table_fixed_slopeseq_netbiaszero
   ```

2. Within the function body, create the table by calling the function `RTW.TflTable`.

   ```
   hTable = RTW.TflTable;
   ```

3. Create the entry for the operator mapping with a call to the `RTW.TflCOperationEntryGenerator` function, which provides access to the fixed-point parameters `SlopesMustBeTheSame` and `MustHaveZeroNetBias`.

   ```
   op_entry = RTW.TflCOperationEntryGenerator;
   ```

4. Set operator entry parameters with a call to the `setTflCOperationEntryParameters` function. The parameters specify the type of operation as addition, the saturation mode as saturation off, rounding modes as unspecified, and the name of the replacement function as `u16_add_SameSlopeZeroBias`. `SlopesMustBeTheSame` and `MustHaveZeroNetBias` are set to `true`, indicating that slopes must be equal and net bias must be zero across the addition inputs and output.

   ```
   setTflCOperationEntryParameters(op_entry, ...
       'Key',                   'RTW_OP_ADD', ...
       'Priority',               90, ...
       'SaturationMode',         'RTW_WRAP_ON_OVERFLOW', ...
       'RoundingModes',          {'RTW_ROUND_UNSPECIFIED'}, ...
       'SlopesMustBeTheSame',    true, ...
       'MustHaveZeroNetBias',    true, ...
       'ImplementationName',      'u16_add_SameSlopeZeroBias', ...
       'ImplementationHeaderFile', 'u16_add_SameSlopeZeroBias.h', ...
       'ImplementationSourceFile', 'u16_add_SameSlopeZeroBias.c');
   ```

5 Create conceptual arguments `y1`, `u1`, and `u2`. There are multiple ways to set up the conceptual arguments. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. Each argument is specified as 16 bits and unsigned. Each argument specifies that code replacement request processing does *not* check for an exact match to the call-site slope and bias values.

```
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);
```

6 Create the implementation arguments. There are multiple ways to set up the implementation arguments. This example uses calls to the `createAndSetCImplementationReturn` and `createAndAddImplementationArg` functions to create and add implementation arguments to the 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`).

```
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);
```

7 Add the entry to a code replacement table with a call to the `addEntry` function.

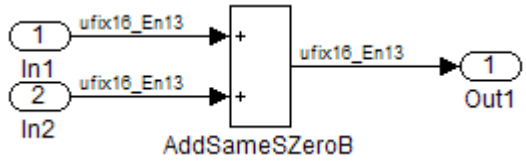
```
addEntry(hTable, op_entry);
```

8 Save the table definition file. Use the name of the table definition function to name the file.

To test this example:

1 Register the code replacement mapping.

2 Create a model.



3 For this model:

- Set the Inport 1 **Data type** to `fixdt(0,16,13)`.
- Set the Inport 2 **Data type** to `fixdt(0,16,13)`.
- In the Add block:
  - Verify that **Output data type** is set to its default, `Inherit via internal rule`.
  - Set **Integer rounding mode** to `Zero`.

4 Configure the model with the following settings:

- On the **Solver** pane, select a fixed-step, discrete solver.
- On the **Code Generation** pane, select an ERT-based system target file.
- On the **Code Generation** > **Interface** pane, select the code replacement library that contains your addition operation entry.

5 Generate code and a code generation report.

6 Review the code replacements.

More About

- "Code You Can Replace From Simulink Models" on page 51-7
- "Define Code Replacement Mappings" on page 51-42
- "Fixed-Point Operator Code Replacement" on page 51-195
- "Data Type Conversions (Casts) and Operator Code Replacement" on page 51-222
- "Shift Left Operations and Code Replacement" on page 51-226
- "Data Alignment for Code Replacement" on page 51-133
- "Remap Operator Output to Function Input" on page 51-192
- "Customize Match and Replacement Process" on page 51-153
- "Develop a Code Replacement Library" on page 51-27

Data Type Conversions (Casts) and Operator Code Replacement

You can use code replacement entries to replace code that the code generator produces for data type conversion (cast) operations.

Casts from `int32` To `int16`

This example creates a code replacement entry that replaces `int32` to `int16` data type conversion (cast) operations. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

1. Create a table definition file that contains a function definition. For example:

   ```
   function hTable = crl_table_cast_int32_to_int16
   ```

2. Within the function body, create the table by calling the function `RTW.TflTable`.

   ```
   hTable = RTW.TflTable;
   ```

3. Create the entry for the operator mapping with a call to the `RTW.TflCOperationEntry` function.

   ```
   op_entry = RTW.TflCOperationEntry;
   ```

4. Set operator entry parameters with a call to the `setTflCOperationEntryParameters` function. The parameters specify the type of operation as cast, the saturation mode as saturate on integer overflow, rounding modes as toward negative infinity, and the name of the replacement function as `my_sat_cast`.

   ```
   setTflCOperationEntryParameters(op_entry, ...
       'Key',                    'RTW_OP_CAST', ...
       'Priority',               50, ...
       'ImplementationName',     'my_sat_cast', ...
       'SaturationMode',         'RTW_SATURATE_ON_OVERFLOW', ...
       'RoundingModes',          {'RTW_ROUND_FLOOR'}, ...
       'ImplementationHeaderFile', 'some_hdr.h', ...
       'ImplementationSourceFile', 'some_hdr.c');
   ```

5. Create the `int16` argument as conceptual argument `y1` and the implementation return value. There are multiple ways to set up the conceptual and implementation arguments. This example uses calls to the `getTflArgFromString` and `addConceptualArg` functions to create the conceptual argument and add it to the entry. Convenience method `setReturn` specifies the argument as the implementation return value.

```
arg = getTflArgFromString(hTable, 'y1', 'int16');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(op_entry, arg);
op_entry.Implementation.setReturn(arg);
```

6 Create the `int32` argument as conceptual and implementation argument `u1`. This example uses calls to the `getTflArgFromString` and `addConceptualArg` functions to create the conceptual argument and add it to the entry. Convenience method `addArgument` specifies the argument as implementation input argument.

```
arg = getTflArgFromString(hTable, 'u1', 'int32');
addConceptualArg(op_entry, arg);
op_entry.Implementation.addArgument(arg);
```

7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hLib, hEnt);
```

8 Save the table definition file. Use the name of the table definition function to name the file.

Casts Using Net Slope

You can use code replacement entries to replace code that the code generator produces for data type conversion (cast) operations.

This example creates a code replacement entry to replace data type conversions (casts) of fixed-point data types by using a net slope. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

1 Create a table definition file that contains a function definition. For example:

```
function hTable = crl_table_cast_fixpt_net_slope
```

2 Within the function body, create the table by calling the function `RTW.TflTable`.

```
hTable = RTW.TflTable;
```

3 Create the entry for the operator mapping with a call to the `RTW.TflCOperationEntryGenerator_Netslope` function, which provides access to the fixed-point parameters `NetSlopeAdjustmentFactor` and `NetFixedExponent`

```
op_entry = RTW.TflCOperationEntryGenerator_NetSlope;
```

4 Set operator entry parameters with a call to the `setTflCOperationEntryParameters` function. The parameters specify the type

of operation as cast, the saturation mode as saturate on integer overflow, rounding modes as toward negative infinity, and the name of the replacement function as `my_fxp_cast`. `NetSlopeAdjustmentFactor` and `NetFixedExponent` specify the `F` and `E` parts of the net slope $F2^E$.

```
InFL = 2;
InWL = 16;
InSgn = true;
OutFL = 4;
OutWL = 32;
OutSgn = true;
setTflCOperationEntryParameters(op_entry, ...
   'Key',                     'RTW_OP_CAST', ...
   'Priority',                50, ...
   'SaturationMode',          'RTW_SATURATE_ON_OVERFLOW', ...
   'RoundingModes',           {'RTW_ROUND_FLOOR'}, ...
   'NetSlopeAdjustmentFactor', 1.0, ...
   'NetFixedExponent',        (OutFL - InFL), ...
   'ImplementationName',      'my_fxp_cast', ...
   'ImplementationHeaderFile', 'some_hdr.h', ...
   'ImplementationSourceFile', 'some_hdr.c');
```

5 Create conceptual arguments `y1` and `u1`. There are multiple ways to set up the conceptual arguments. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. Each argument is specified as fixed-point and signed. Each argument specifies that code replacement request processing does *not* check for an exact match to the call-site slope and bias values.

```
createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
    'Name',         'y1', ...
    'IOType',       'RTW_IO_OUTPUT', ...
    'CheckSlope',   false, ...
    'CheckBias',    false, ...
    'DataTypeMode', 'Fixed-point: binary point scaling', ...
    'IsSigned',     OutSgn, ...
    'WordLength',   OutWL, ...
    'FractionLength',OutFL);

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
     'Name',         'u1', ...
     'IOType',       'RTW_IO_INPUT', ...
     'CheckSlope',   false, ...
     'CheckBias',    false, ...
     'DataTypeMode', 'Fixed-point: binary point scaling', ...
     'IsSigned',     InSgn, ...
     'WordLength',   InWL, ...
     'FractionLength',InFL);
```

6 Create the implementation arguments. There are multiple ways to set up the implementation arguments. This example uses calls to the `createAndSetCImplementationReturn` and

`createAndAddImplementationArg` functions to create and add implementation arguments to the entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types).

```
createAndSetCImplementationReturn(op_entry, 'RTW.TflArgNumeric', ...
    'Name',           'y1', ...
    'IOType',         'RTW_IO_OUTPUT', ...
    'IsSigned',       OutSgn, ...
    'WordLength',     OutWL, ...
    'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
    'Name',           'u1', ...
    'IOType',         'RTW_IO_INPUT', ...
    'IsSigned',       InSgn, ...
    'WordLength',     InWL, ...
    'FractionLength', 0);
```

7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

8 Save the table definition file. Use the name of the table definition function to name the file.

More About

- “Code You Can Replace From Simulink Models” on page 51-7
- “Define Code Replacement Mappings” on page 51-42
- “Fixed-Point Operator Code Replacement” on page 51-195
- “Shift Left Operations and Code Replacement” on page 51-226
- “Data Alignment for Code Replacement” on page 51-133
- “Remap Operator Output to Function Input” on page 51-192
- “Customize Match and Replacement Process” on page 51-153
- “Develop a Code Replacement Library” on page 51-27

Shift Left Operations and Code Replacement

You can use code replacement entries to replace code that the code generator produces for shift (<<) operations.

Shift Lefts for `int16` Data

This example creates a code replacement entry to replace shift left operations for `int16` data. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

1. Create a table definition file that contains a function definition. For example:

   ```
   function hTable = crl_table_shift_left_int16
   ```

2. Within the function body, create the table by calling the function `RTW.TflTable`.

   ```
   hTable = RTW.TflTable;
   ```

3. Create the entry for the operator mapping with a call to the `RTW.TflCOperationEntry` function.

   ```
   op_entry = RTW.TflCOperationEntry;
   ```

4. Set operator entry parameters with a call to the `setTflCOperationEntryParameters` function. The parameters specify the type of operation as shift left and the name of the replacement function as `my_shift_left`.

   ```
   setTflCOperationEntryParameters(op_entry, ...
       'Key',                    'RTW_OP_SL', ...
       'Priority',               50, ...
       'ImplementationName',     'my_shift_left', ...
       'ImplementationHeaderFile', 'some_hdr.h', ...
       'ImplementationSourceFile', 'some_hdr.c');
   ```

5. Create the `int16` argument as conceptual argument `y1` and the implementation return value. There are multiple ways to set up the conceptual and implementation arguments. This example uses calls to the `getTflArgFromString` and `addConceptualArg` functions to create the conceptual argument and add it to the entry. Convenience method `setReturn` specifies the argument as the implementation return value.

   ```
   arg = getTflArgFromString(hTable, 'y1', 'int16');
   arg.IOType = 'RTW_IO_OUTPUT';
   addConceptualArg(op_entry, arg);
   op_entry.Implementation.setReturn(arg);
   ```

6 Create the `int16` argument as conceptual and implementation argument `u1`. This example uses calls to the `getTflArgFromString` and `addConceptualArg` functions to create the conceptual argument and add it to the entry. Convenience method `addArgument` specifies the argument as an implementation input argument.

```
arg = getTflArgFromString(hTable, 'u1', 'int16');
addConceptualArg(op_entry, arg);
op_entry.Implementation.addArgument(arg);
```

7 Create the `int8` argument as conceptual and implementation argument `u2`. This example uses calls to the `getTflArgFromString` and `addConceptualArg` functions to create the conceptual argument and add it to the entry. This argument specifies the number of bits to shift the previous input argument. Because the argument type is not relevant, the example disables type checking by setting the `CheckType` property to `false`. Convenience method `addArgument` specifies the argument as implementation input argument.

```
arg = getTflArgFromString(hTable, 'u2', 'int8');
arg.CheckType = false;
addConceptualArg(op_entry, arg);
op_entry.Implementation.addArgument(arg);
```

- The function `getTflArgFromString` is called to create an `int8` input argument. This argument is added to the operator entry both as the third conceptual argument and the second implementation input argument.
- Add the entry to a code replacement table with a call to the `addEntry` function.

  ```
  addEntry(hTable, op_entry);
  ```
- Save the table definition file. Use the name of the table definition function to name the file.

Shift Lefts Using Net Slope

You can use code replacement entries to replace code that the code generator produces for shift (<<) operations.

This example creates a code replacement entry to replace shift left operations for fixed-point data using a net slope. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

1 Create a table definition file that contains a function definition. For example:

```
function hTable = crl_table_shift_left_fixpt_net_slope
```

2 Within the function body, create the table by calling the function `RTW.TflTable`.

```
hTable = RTW.TflTable;
```

3 Create the entry for the operator mapping with a call to the `RTW.TflCOperationEntryGenerator_Netslope` function. This function provides access to the fixed-point parameters `NetSlopeAdjustmentFactor` and `NetFixedExponent`.

```
op_entry = RTW.TflCOperationEntryGenerator_NetSlope;
```

4 Set operator entry parameters with a call to the `setTflCOperationEntryParameters` function. The parameters specify the type of operation as shift left, the saturation mode as saturate on integer overflow, rounding modes as toward negative infinity, and the name of the replacement function as `my_fxp_shift_left`. `NetSlopeAdjustmentFactor` and `NetFixedExponent` specify the `F` and `E` parts of the net slope $F2^E$.

```
InFL = 2;
InWL = 16;
InSgn = true;
OutFL = 4;
OutWL = 32;
OutSgn = true;
setTflCOperationEntryParameters(op_entry, ...
   'Key',                      'RTW_OP_SL', ...
   'Priority',                 50, ...
   'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
   'RoundingModes',            {'RTW_ROUND_FLOOR'}, ...
   'NetSlopeAdjustmentFactor', 1.0, ...
   'NetFixedExponent',         (OutFL - InFL),...
   'ImplementationName',       'my_fxp_shift_left', ...
   'ImplementationHeaderFile', 'some_hdr.h', ...
   'ImplementationSourceFile', 'some_hdr.c');
```

5 Create conceptual arguments `y1` and `u1`. There are multiple ways to set up the conceptual arguments. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. Each argument is specified as fixed-point and signed. Each argument specifies that code replacement request processing does *not* check for an exact match to the call-site slope and bias values.

```
createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
    'Name',         'y1', ...
    'IOType',       'RTW_IO_OUTPUT', ...
    'CheckSlope',   false, ...
    'CheckBias',    false, ...
    'DataTypeMode', 'Fixed-point: binary point scaling', ...
    'IsSigned',     OutSgn, ...
    'WordLength',   OutWL, ...
```

```
    'FractionLength',OutFL);

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
    'Name',          'u1', ...
    'IOType',        'RTW_IO_INPUT', ...
    'CheckSlope',    false, ...
    'CheckBias',     false, ...
    'DataTypeMode', 'Fixed-point: binary point scaling', ...
    'IsSigned',      InSgn, ...
    'WordLength',    InWL, ...
    'FractionLength',InFL);
```

6 Create the implementation arguments. There are multiple ways to set up the implementation arguments. This example uses calls to the `createAndSetCImplementationReturn` and `createAndAddImplementationArg` functions to create and add implementation arguments to the entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types).

```
createAndSetCImplementationReturn(op_entry, 'RTW.TflArgNumeric', ...
    'Name',           'y1', ...
    'IOType',         'RTW_IO_OUTPUT', ...
    'IsSigned',       OutSgn, ...
    'WordLength',     OutWL, ...
    'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric', ...
     'Name',           'u1', ...
     'IOType',         'RTW_IO_INPUT', ...
     'IsSigned',       InSgn, ...
     'WordLength',     InWL, ...
     'FractionLength', 0);
```

7 Create the `int8` argument as conceptual and implementation argument `u2`. This example uses calls to the `getTflArgFromString` and `addConceptualArg` functions to create the conceptual argument and add it to the entry. This argument specifies the number of bits to shift the previous input argument. Because the argument type is not relevant, type checking is disabled by setting the `CheckType` property to `false`. Convenience method `addArgument` specifies the argument as implementation input argument.

```
arg = getTflArgFromString(hTable, 'u2', 'uint8');
arg.CheckType = false;
addConceptualArg(op_entry, arg);
op_entry.Implementation.addArgument(arg);
```

8 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

9 Save the table definition file. Use the name of the table definition function to name the file.

More About

- “Code You Can Replace From Simulink Models” on page 51-7
- “Define Code Replacement Mappings” on page 51-42
- “Fixed-Point Operator Code Replacement” on page 51-195
- “Data Type Conversions (Casts) and Operator Code Replacement” on page 51-222
- “Data Alignment for Code Replacement” on page 51-133
- “Remap Operator Output to Function Input” on page 51-192
- “Customize Match and Replacement Process” on page 51-153
- “Develop a Code Replacement Library” on page 51-27

Code Replacement Customization for MATLAB Code

What Is Code Replacement Customization?

Customize how and when the code generator replaces C/C++ code that it generates by default for functions and operators by developing a custom code replacement library. You can develop libraries interactively with the Code Replacement Tool or programmatically.

- Develop libraries tailored to specific application requirements
- Add identifiers to the list of reserved keywords the code generator considers during code replacement
- Customize the code generator's match and replacement process for functions

To get started, "Quick Start Library Development" on page 51-28.

Code Replacement Match and Replacement Process

When the code generator encounters a call site for a function or operator, it:

1. Creates and partially populates a code replacement entry object with the function or operator name or key and conceptual arguments.
2. Uses the entry object to query the configured code replacement library for a conceptual representation match. The code generator searches the tables in a code replacement library for a match in the order that the tables appear in the library. When searching for a match, the code generator takes into account:
   - Conceptual name or key
   - Arguments, including quantity, type, type qualifiers, and complexity
   - Algorithm (computation method)
   - Fixed-point saturation and rounding modes
   - Priority
3. When a match exists, the code generator returns a code replacement object, fully populated with the conceptual representation, implementation representation, and priority. If the code generator finds multiple matches within a table, the entry priority determines the match. The priority can range from 0 to 100. The highest priority is 0. The code generator uses a higher-priority entry over a similar entry with a lower priority.
4. Uses the C or C++ replacement function prototype in the code replacement object to generate code.

Code Replacement Customization Limitations

- Code replacement verification — It is possible that code replacement behaves differently than you expect. For example, data types that you observe in code generator input might not match what the code generator uses as intermediate data types during an operation. Verify code replacements by examining generated code. See “Verify Code Replacements” on page 52-64.
- Tokens in file paths—You can include tokens in file paths when specifying build information for a code replacement entry by using the programming interface only. The ability to include tokens is not available from the Code Replacement Tool. See “Specify Build Information for Replacement Code” on page 52-47.
- Addition and subtraction operation replacements—See “Addition and Subtraction Operator Code Replacement” on page 52-122for relevant limitations.
- `coder.replace` function — See `coder.replace` for relevant limitations.
- `coder.dataAlignment` function — See `coder.dataAlignment` for relevant limitations.

More About

- “Code You Can Replace from MATLAB Code” on page 52-5
- “Develop a Code Replacement Library” on page 52-15
- “Quick Start Library Development” on page 52-16
- “What Is Code Replacement?” (MATLAB Coder)

Code You Can Replace from MATLAB Code

Code that the code generator replaces depends on the code replacement library (CRL) that you use. By default, the code generator does not apply a code replacement library. Your choice of libraries is dependent on product licensing and whether you have access to custom libraries.

In this section...

Math Functions

When generating C/C++ code from MATLAB code, depending on code replacement libraries available in your development environment, you can configure the code generator to replace instances of the following math functions with application-specific implementations.

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
`abs`[1]	Floating point	Scalar	Real
`acos`	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
`acosd`	Floating point	Scalar Vector Matrix	Real Complex
`acot`	Floating point	Scalar Vector Matrix	Real Complex
`acotd`	Floating point	Scalar Vector Matrix	Real Complex
`acoth`	Floating point	Scalar	Real

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
		Vector Matrix	Complex
`acsc`	Floating point	Scalar Vector Matrix	Real Complex
`acscd`	Floating point	Scalar Vector Matrix	Real Complex
`acsch`	Floating point	Scalar Vector Matrix	Real Complex
`asec`	Floating point	Scalar Vector Matrix	Real Complex
`asecd`	Floating point	Scalar Vector Matrix	Real Complex
`asech`	Floating point	Scalar Vector Matrix	Real Complex
`asin`	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
`asind`	Floating point	Scalar Vector Matrix	Real Complex
`atan`	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
`atan2`	Floating point	Scalar Vector Matrix	Real
`atan2d`	Floating point	Scalar Vector Matrix	Real
`atand`	Floating point	Scalar Vector Matrix	Real Complex
`cos`	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
`ceil`	• Floating-point • Scalar	• Floating-point • Scalar	• Floating-point • Scalar
`cosd`	Floating point	Scalar Vector Matrix	Real Complex
`cosh`	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
`cot`	Floating point	Scalar Vector Matrix	Real Complex
`cotd`	Floating point	Scalar Vector Matrix	Real Complex
`coth`	Floating point	Scalar Vector Matrix	Real Complex

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
`csc`	Floating point	Scalar Vector Matrix	Real Complex
`cscd`	Floating point	Scalar Vector Matrix	Real Complex
`csch`	Floating point	Scalar Vector Matrix	Real Complex
`exp`	Floating point	Scalar	Real
`fix`	Floating point	Scalar	Real
`floor`	• Floating-point • Scalar	• Floating-point • Scalar	• Floating-point • Scalar
`hypot`	Floating point	Scalar Vector Matrix	Real
`ldexp`	Floating point	Scalar	Real
`log`	Floating point	Scalar Vector Matrix	Real Complex
`log10`	Floating point	Scalar Vector Matrix	Real Complex
`log2`	Floating point	Scalar Vector Matrix	Real Complex
`max`	Integer Floating point	Scalar	Real
`min`	Integer Floating point	Scalar	Real
`pow`	Floating point	Scalar	Real

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
`rem`	Floating point	Scalar	Real
`round`	Floating point	Scalar	Real
`sec`	Floating point	Scalar Vector Matrix	Real Complex
`secd`	Floating point	Scalar Vector Matrix	Real Complex
`sech`	Floating point	Scalar Vector Matrix	Real Complex
`sign`	Floating point	Scalar	Real
`sin`	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
`sind`	Floating point	Scalar Vector Matrix	Real Complex
`sinh`	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
`sqrt`	Floating point	Scalar	Real
`tan`	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
`tand`	Floating point	Scalar Vector Matrix	Real Complex

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
`tanh`	Floating point	Scalar Vector Matrix	Real Complex Complex input/complex output Real input/complex output
[1] Wrap on integer overflow only			

Memory Functions

Depending on code replacement libraries available in your development environment, you can configure the code generator to replace instances of the following memory functions with application-specific implementations.

Function	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
`memcmp`	Void pointer (`void*`)	Scalar Vector Matrix	Real Complex
`memcpy`	Void pointer (`void*`)	Scalar Vector Matrix	Real Complex
`memset`	Void pointer (`void*`)	Scalar Vector Matrix	Real Complex
`memset2zero`	Void pointer (`void*`)	Scalar Vector Matrix	Real Complex

Some target processors provide optimized functions to set memory to zero. Use the code replacement library programming interface to replace the `memset2zero` function with more efficient target-specific functions.

Operators

When generating C/C++ code from MATLAB code, depending on code replacement libraries available in your development environment, you can configure the code

generator to replace instances of the following operators with application-specific implementations.

Mixed data type support indicates you can specify different data types of different inputs.

Operator	Key	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
Addition (+)[1]	`RTW_OP_ADD`	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Subtraction (-)[1]	`RTW_OP_MINUS`	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Multiplication (*)[2]	`RTW_OP_MUL`	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Division (/)	`RTW_OP_DIV`	Integer Floating point Fixed-point Mixed	Scalar	Real Complex
Data type conversion (cast)	`RTW_OP_CAST`	Integer Floating point[3] Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Shift left (<<)	`RTW_OP_SL`	Integer Fixed-point Mixed	Scalar Vector Matrix	Real
Shift right arithmetic (>>)[4]	`RTW_OP_SRA`	Integer Fixed-point Mixed	Scalar Vector Matrix	Real
Shift right logical (>>)	`RTW_OP_SRL`	Integer Fixed-point	Scalar Vector	Real

Operator	Key	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
		Mixed	Matrix	
Element-wise matrix multiplication (`.*`)[5]	`RTW_OP_ELEM_MUL`	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Complex conjugation	`RTW_OP_CONJUGATE`	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Transposition (`.'`)	`RTW_OP_TRANS`	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Hermitian (complex conjugate) transposition (`'`)	`RTW_OP_HERMITIAN`	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Multiplication with transposition[2]	`RTW_OP_TRMUL`	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Multiplication with Hermitian transposition[2]	`RTW_OP_HMMUL`	Integer Floating point Fixed-point Mixed	Vector Matrix	Real Complex
Multiplication followed by shift right arithmetic (`u1*u2>>u3`)[6]	`RTW_OP_MUL_SRA`	Integer Fixed-point	Scalar	Real
Multiplication followed by division (`u1*u2/u3`)[7]	`RTW_OP_MULDIV`	Integer Fixed-point	Scalar	Real

Operator	Key	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support
Greater than (>)	RTW_OP_GREATER_THAN	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Greater than or equal (>=)	RTW_OP_GREATER_THAN_OR_EQUAL	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Less than (<)	RTW_OP_LESS_THAN	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Less than or equal (<=)	RTW_OP_LESS_THAN_OR_EUQAL	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Equal (==)	RTW_OP_EUQAL	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex
Not equal (!=)	RTW_OP_NOT_EUQAL	Integer Floating point Fixed-point Mixed	Scalar Vector Matrix	Real Complex

Operator	Key	Data Type Support	Scalar, Vector, Matrix Support	Real, Complex Support

[1] See "Addition and Subtraction Operator Code Replacement" on page 52-122 for details to consider when defining mappings for addition and subtraction code replacements.

[2] Can map to Basic Linear Algebra Subroutine (BLAS) multiplication functions.

[3] Scaled floating point is not supported.

[4] Code replacement libraries that provide arithmetic shift right implementations should also provide logical shift right implementations, because some arithmetic shift rights are converted to logical shift rights during code generation.

[5] Use the multiplication (`*`) operator (`RTW_OP_MUL`) for scalar multiplication.

[6] Requires scalar, real, or fixed-point data types with zero bias; output type of the multiplication operation to accommodate all possible output values; shift operand is an unsigned integer; and net slope is equal to 1 (`U1_slope * U2_slope == Mul_output_slope` and `Mul_output_slope == output_slope_of_shift_operation`).

[7] Requires scalar, real, or fixed-point data types with zero bias; output type of the multiplication operation to accommodate all possible output values; and net slope is equal to 1 (`U1_slope * U2_slope == Mul_output_slope == U3_slope * Div_output_slope`).

More About

- "Develop a Code Replacement Library" on page 52-15
- "Quick Start Library Development" on page 52-16
- "What Is Code Replacement?" (MATLAB Coder)

Develop a Code Replacement Library

Iterate through the following steps, as necessary, to develop a code replacement library:

1. "Identify Code Replacement Requirements" on page 52-26
2. "Prepare for Code Replacement Library Development" on page 52-29
3. "Define Code Replacement Mappings" on page 52-30
4. "Specify Build Information for Replacement Code" on page 52-47
5. "Register Code Replacement Mappings" on page 52-56
6. "Verify Code Replacements" on page 52-64
7. "Deploy Code Replacement Library" on page 52-81

To get started, see "Identify Code Replacement Requirements" on page 52-26.

To experiment with the process and tools, see "Quick Start Library Development" on page 52-16.

More About

- "Identify Code Replacement Requirements" on page 52-26
- "Code You Can Replace from MATLAB Code" on page 52-5
- "Quick Start Library Development" on page 52-16
- "What Is Code Replacement Customization?" on page 52-3

Quick Start Library Development

This example shows how to develop a code replacement library that includes an entry for generating replacement code for the math function `sin`. You use the Code Replacement Tool.

Prerequisites

To complete this example, install the following software:

- MATLAB
- MATLAB Coder
- Embedded Coder
- C compiler

For instructions on installing MathWorks products, see "Installation and Activation" (Installation, Licensing, and Activation). If you have installed MATLAB and want to see what other MathWorks products are installed, in the Command Window, enter `ver`.

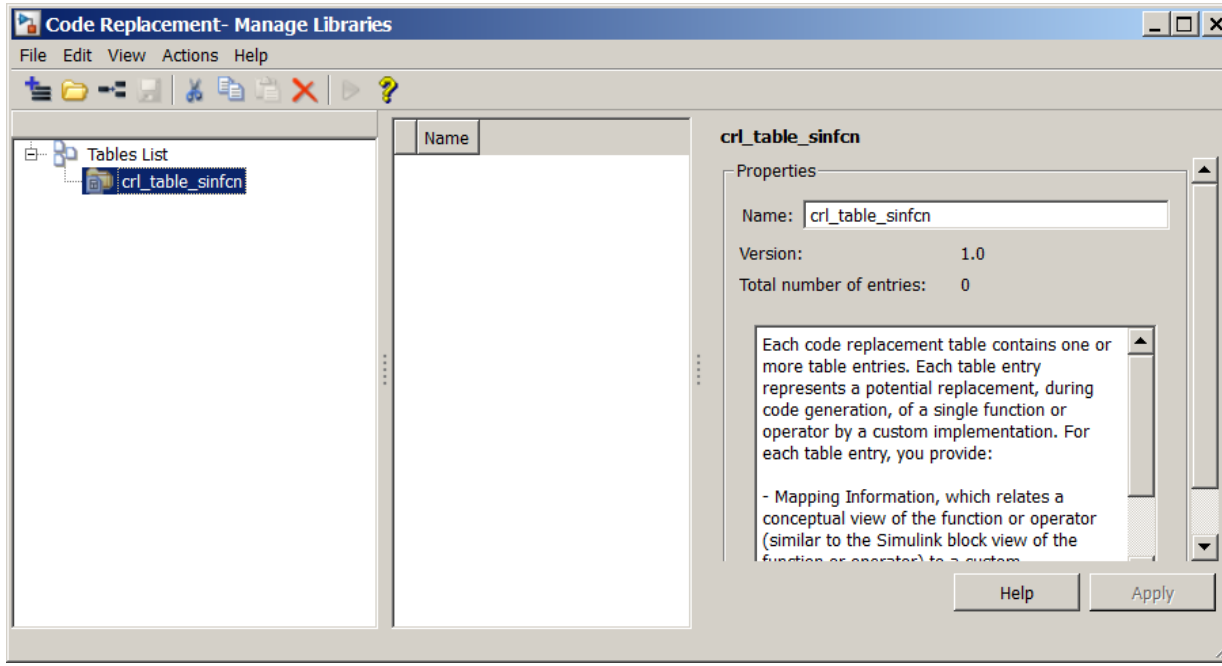
For a list of supported compilers, see http://www.mathworks.com/support/compilers/current_release/.

Open the Code Replacement Tool

1. Start a new MATLAB session.
2. Create or navigate (`cd`) to an empty folder.
3. At the command prompt, enter the `crtool` command. The Code Replacement Tool window opens.

Create Code Replacement Table

1. In the Code Replacement Tool window, select **File > New table**.
2. In the right pane, name the table `crl_table_sinfcn` and click **Apply**. Later, when you save the table, the tool saves it with the file name `crl_table_sinfcn.m`.



Create Table Entry

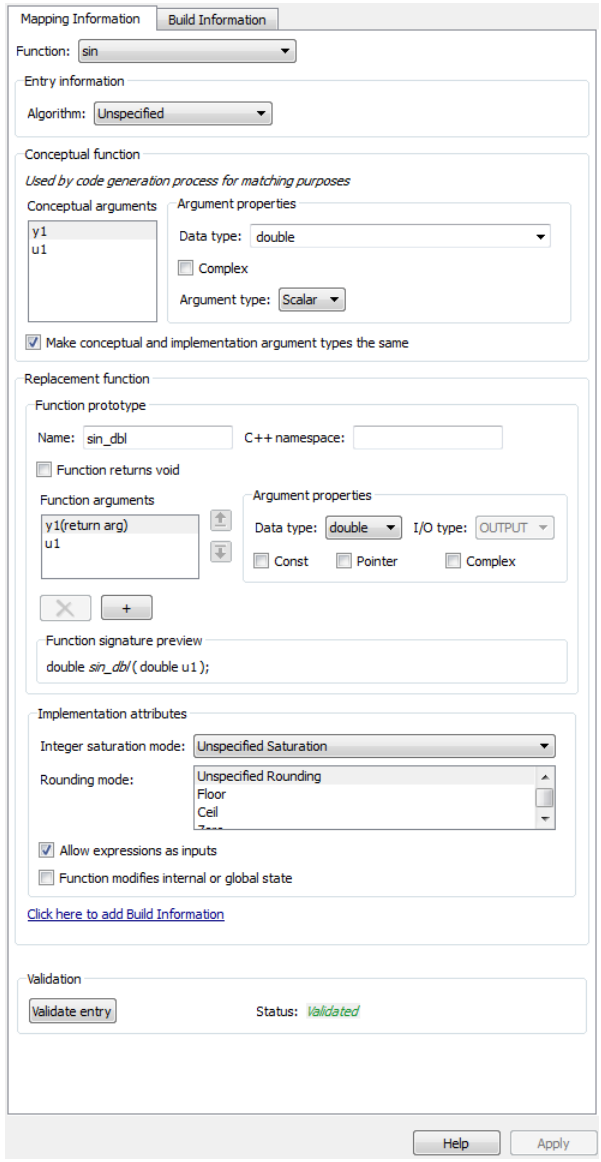
Create a table entry that maps a `sin` function with `double` input and `double` output to a custom implementation function.

1. In the left pane, select table `crl_table_sinfcn`. Then, select **File** > **New entry** > **Function**. The new entry appears in the middle pane, initially without a name.
2. In the middle pane, select the new entry.
3. In the right pane, on the **Mapping Information** tab, from the **Function** menu, select `sin`.
4. Leave **Algorithm** set to `Unspecified`, and leave parameters in the **Conceptual function** group set to default values.
5. In the **Replacement function** group, name the replacement function `sin_dbl`.
6. Leave the remaining parameters in the **Replacement function** group set to default values.

7 Click **Apply**. The tool updates the **Function signature preview** to reflect the specified replacement function name.

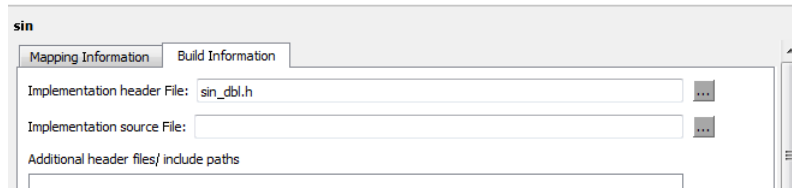
8 Scroll to the bottom of the **Mapping Information** tab and click **Validate entry**. The tool validates your entry.

The following figure shows the completed mapping information.


Mapping Information
Build Information
Function: sin
Entry information
Algorithm: Unspecified
Conceptual function
Used by code generation process for matching purposes
Conceptual arguments
y1
u1
Argument properties
Data type: double
Complex
Argument type: Scalar
Make conceptual and implementation argument types the same
Replacement function
Function prototype
Name: sin_dbl
C++ namespace:
Function returns void
Function arguments
y1(return arg)
u1
Argument properties
Data type: double
I/O type: OUTPUT
Const
Pointer
Complex
Function signature preview
double sin_dbl(double u1);
Implementation attributes
Integer saturation mode: Unspecified Saturation
Rounding mode:
Unspecified Rounding
Floor
Ceil
Allow expressions as inputs
Function modifies internal or global state
Click here to add Build Information
Validation
Validate entry
Status: Validated
Help
Apply

Specify Build Information for Replacement Code

1. On the **Build Information** tab, for the **Implementation header file** parameter, enter `sin_dbl.h`.
2. Leave the remaining parameters set to default values.
3. Click **Apply**.



4. Optionally, you can revalidate the entry. Return to the **Mapping Information** tab and click **Validate entry**.

Create Another Table Entry

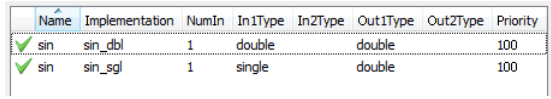
Create an entry that maps a `sin` function with `single` input and `double` output to a custom implementation function named `sin_sgl`. Create the entry by copying and pasting the `sin_dbl` entry.

1. In the middle pane, select the `sin_dbl` entry.
2. Select **Edit** > **Copy**
3. Select **Edit** > **Paste**
4. On the **Mapping Information** tab, in the **Conceptual function** section, set the data type of input argument `u1` to `single`.
5. In the **Replacement function** section, name the function `sin_sgl`. Set the data type of input argument `u1` to `single`.
6. Click **Apply**. Note the changes that appear for the **Function signature preview**.
7. On the **Build Information** tab, for the **Implementation header file** parameter, enter `sin_sgl.h`. Leave the remaining parameters set to default values and click **Apply**.

Validate the Code Replacement Table

1. Select **Actions** > **Validate table**.

2 If the tool reports errors, fix them, and rerun the validation. Repeat fixing and validating errors until the tool does not report errors. The following figure shows a validation report.



	Name	Implementation	NumIn	In1Type	In2Type	Out1Type	Out2Type	Priority
✓	sin	sin_dbl	1	double		double		100
✓	sin	sin_sgl	1	single		double		100

Save the Code Replacement Table

Save the code replacement table to a MATLAB file in your working folder. Select **File** > **Save table**. By default, the tool uses the table name to name the file. For this example, the tool saves the table in the file `crl_table_sinfcn.m`.

Review the Code Replacement Table Definition

Consider reviewing the MATLAB code for your code replacement table definition. After using the tool to create an initial version of a table definition file, you can update, enhance, or copy the file in a text editor.

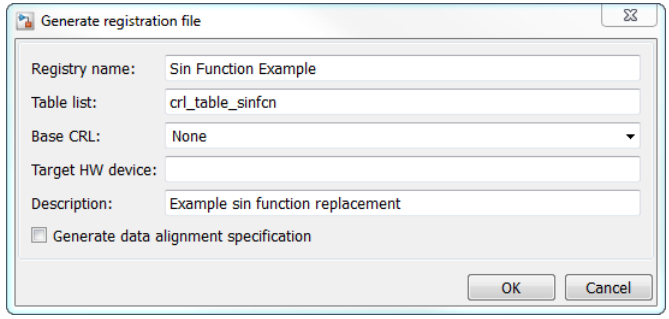
To review it, in MATLAB or another text editor, open the file `crl_table_sinfcn.m`.

Generate a Registration File

Before you can use your code replacement table, you must register it as part of a code replacement library. Use the Code Replacement Tool to generate a registration file.

1 In the Code Replacement Tool, select **File** > **Generate registration file**.

2 In the **Generate registration file** dialog box, edit the dialog box fields to match the following figure, and then click **OK**.



3 In the **Select location** dialog box, specify a location for the registration file. The location must be on the MATLAB path or in the current working folder. Save the file. The tool saves the file as `rtwTargetInfo.m`.

Register the Code Replacement Table

At the command prompt, enter:

```
RTW.TargetRegistry.getInstance('reset');
```

Review and Test Code Replacements

Apply your code replacement library. Verify that the code generator makes code replacements that you expect.

1 Check for errors. At the command line, invoke the table definition file . For example:

```
tbl = crl_table_sinfcn

tbl =

  TflTable with properties:


               Version: '1.0'
       ReservedSymbols: []
    StringResolutionMap: []
             AllEntries: [2x1 RTW.TflCFunctionEntry]
           EnableTrace: 1
```

If an error exists in the definition file, the invocation triggers a message to appear. Fix the error and try again.

2 Use the Code Replacement Viewer to check your code replacement entries. For example:

```
crviewer('Sin Function Example')
```

In the viewer, select entries in your table and verify that the content is what you expect. The viewer can help you detect issues such as:

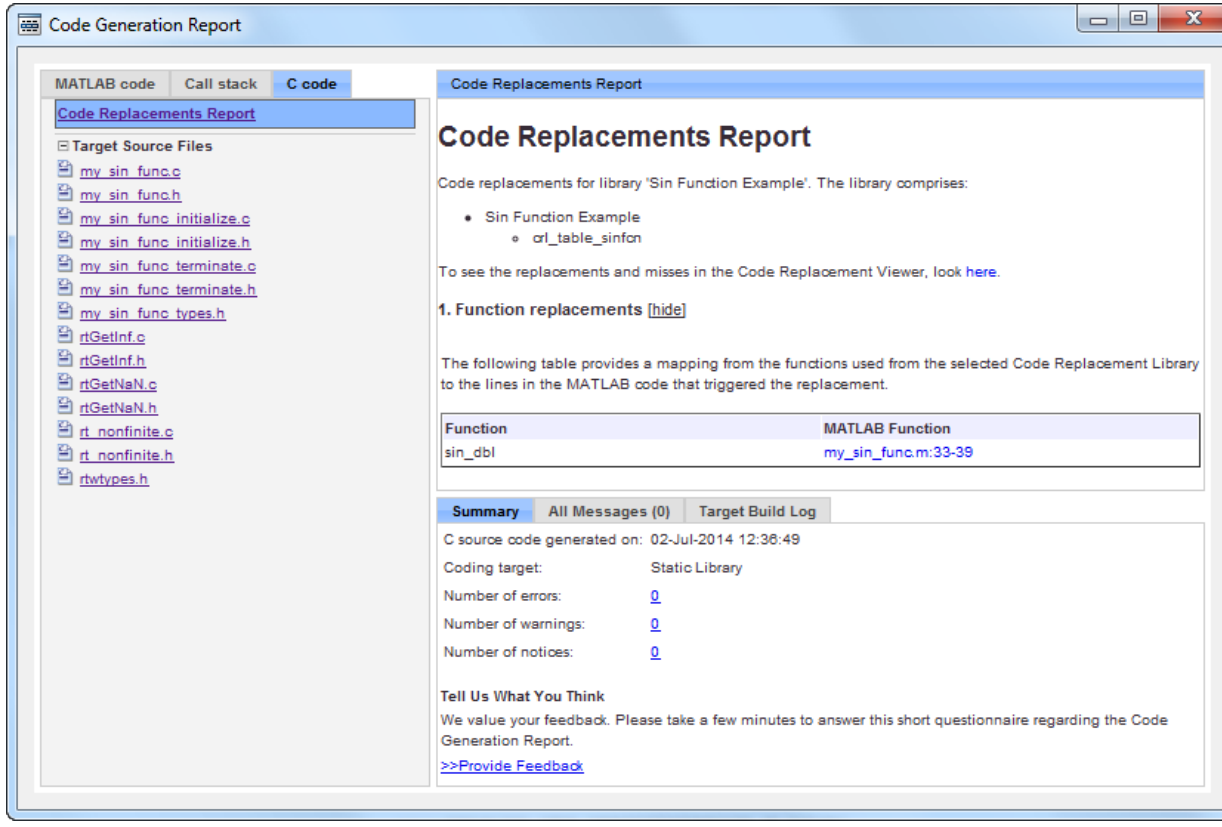
- Incorrect argument order.
- Conceptual argument names that do not match what is expected by the code generator.

- Incorrect priority settings.

3. Identify existing or create new MATLAB code that calls the `sin` function. For example:

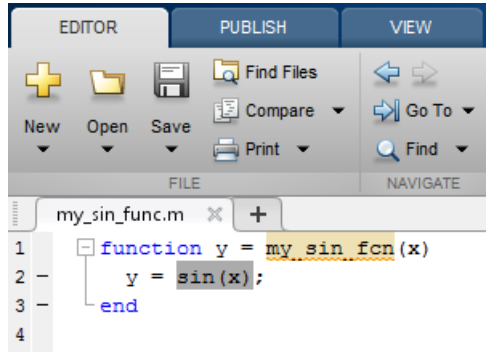
```
function y = my_sin_fnc(x)
  y = sin(x);
end
```

4. Open the MATLAB Coder app.
5. Add the function that includes a call to the `sin` function as an entry-point file. For example, add `my_sin_func.m`. The app creates a project named `my_sin_func.prj`.
6. Click **Next** to go to the **Define Input Type** step. Define the types for the entry-point function inputs.
7. Click **Next** to go to the **Check for Run-Time Issues** step. This step is optional. However, it is a best practice to perform this step. Provide a test file that calls your entry-point function. The app generates a MEX function from your entry-point function. Then, the app runs the test file, replacing calls to the MATLAB function with calls to the generated MEX function.
8. Click **Next** to go to the **Generate Code** step. To open the **Generate** dialog box, click the **Generate** arrow  .
9. Set **Build type** to generate a library or executable.
10. Click **More Settings**.
11. Configure the code generator to use your code replacement library. On the **Custom Code** tab, set the **Code replacement library** parameter to the name of your library. For example, `Sin Function Example`.
12. Configure the code generation report. On the **Debugging** tab, set the **Always create a code generation report**, **Code replacements**, and **Automatically launch a report if one is generated** parameters.
13. Configure the code generator to generate code only. On the **Generate** dialog box, select the **Generate code only** check box. You want to review your code replacements in the generated code before building an executable.
14. Click **Generate** to generate C code and a report.
15. Review code replacement results in the Code Replacements Report section of the code generation report.



The report indicates that the code generator found a match and applied the replacement code for the function `sin_dbl`.

16 Review the code replacements. In the report, under **Function replacements**, click the MATLAB function that triggered the replacement, `my_sin_func.m`. The MATLAB Editor opens and highlights the function call that triggers the code replacement.



More About

- "Develop a Code Replacement Library" on page 52-15
- "What Is Code Replacement Customization?" on page 52-3

Identify Code Replacement Requirements

The first step to developing a code replacement library is to consider the following types of requirements for your code replacement library.

Mapping Information Requirements

- Are you defining a code replacement mapping for the first time?
- Are you updating code replacement entries in an existing library? Or, are you creating a new library?
- Are you rapid prototyping code replacements?
- Can you base your mappings on existing mappings?
- What type of code do you want to replace? Options include:
  - Math operation
  - Function
  - BLAS operation
  - CBLAS operation
  - Net slope fixed-point operation
  - Semaphore or mutex functions
- Do you want to change the inline or nonfinite behavior for functions?
- What specific functions and operations do you want to replace?
- What input and output arguments does the function or operator that you are replacing take? For each argument, what is the data type, complexity, and dimensionality?
- What does the prototype for your replacement code look like?
  - What is the replacement function name?
  - What are the input and output arguments?
  - Are there return values?
  - What is the data type, complexity, and dimensionality of each argument and return value?

Build Information Requirements

- Does your replacement function implementation require a header file? If yes, specify the header file.
- If the replacement function implementation requires a header file, what is the path for that file?
- Is the source file for the replacement function in your working folder? If not, you can explicitly specify the source file name and extension. For example, if the file is required in the generated makefile or specified in a build information object, specify the source file.
- Does the replacement function use additional include files? If yes, what are they and what are the paths for those files?
- Does the replacement function use additional source files? If yes, what are they and what are the paths for those files?
- What compiler flags are required for compiling code that includes the replacement code?
- What linker flags are required for building an executable that includes the replacement code?
- Are the required header, source, and object files for building an executable that includes your replacement code in the working folder for your project? If not, before starting the build process, do you want the code generator to copy required files to the build folder?

Registration Information Requirements

- What do you want to name your code replacement library?
- What code replacement tables do you want to include in the library? What are the file names and paths for the tables?
- What is the purpose of the library? You can document the purpose as the library description.
- Does the library apply to specific hardware devices? If yes, what devices?
- Are you developing a hierarchy of code replacement libraries? Is the library that you are developing based (dependent) on another library? For example, you can specify a general `TI device library` as the base library for a more specific `TI C28x` device library.

- Do you need to specify data alignment for the library? What data alignments are required? For each specification, what type of alignment is required and for what programming language?

Next, prepare for developing a library by reviewing a code replacement library development checklist.

Related Examples

- “Develop a Code Replacement Library” on page 52-15
- “Prepare for Code Replacement Library Development” on page 52-29
- “What Is Code Replacement Customization?” on page 52-3
- “Code You Can Replace from MATLAB Code” on page 52-5

Prepare for Code Replacement Library Development

After you identify your code replacement requirements, prepare for library development by reviewing this checklist:

- Get familiar with the library development process.
- Decide whether to define code replacement mappings and produce a registration file interactively with the Code Replacement Tool or programmatically.
- Identify or develop MATLAB code and Simulink models to test your code replacement library.
- Consider the hierarchy and organization of your library. A library can consist of multiple tables and each table can include multiple entries. How do you want to organize the library to optimize reuse of tables and entries? For example, a registration file can define code replacement tables organized in a hierarchy of code replacement libraries based on entries that increase in specificity:
  - Common entries
  - Entries for TI devices
  - Entries for TI C6xx devices
  - Entries specific to the TI C67x device
- If support files, such as header files, additional source files, and dynamically linked libraries are not in your current working folder, note their location. You need to specify the paths for such files.

Next, based on your requirements and preparation, define code replacement mappings.

More About

- “Identify Code Replacement Requirements” on page 52-26
- “Code You Can Replace from MATLAB Code” on page 52-5
- “Define Code Replacement Mappings” on page 52-30
- “Develop a Code Replacement Library” on page 52-15
- “What Is Code Replacement Customization?” on page 52-3

Define Code Replacement Mappings

After you prepare for library development, use your requirements to define code replacement mappings. A code replacement mapping associates a conceptual representation of a function or operator that is familiar to the code generator with a custom implementation representation that specifies a C or C++ replacement function prototype. You capture a mapping as an entry in a code replacement table:

- Interactively, by using the Code Replacement Tool.
- Programmatically, by using a MATLAB programming interface.

Choose an Approach for Defining Code Replacement Mappings

The following table lists situations to help you decide when to use the interactive or programmatic approach.

Situation	Approach
Defining mappings for the first time.	Code Replacement Tool.
Rapid prototyping mappings.	Code Replacement Tool to quickly generate, register, and test mappings.
Developing a mapping as a template or starting point for defining similar mappings.	Code Replacement Tool to generate definition code that you can copy and modify.
Modifying a registration file, including copying and pasting content.	MATLAB Editor to update the programming interface directly.
Defining mappings that specify attributes not available from the Code Replacement Tool (for example, sets of algorithm parameters).	Programming interface.
Reusing existing code for new mappings by copying, pasting, and editing existing mappings.	Programming interface.

Define Mappings Interactively with the Code Replacement Tool

This example shows how to use the Code Replacement Tool to develop code replacement mappings. The tool is ideal for getting started with developing mappings, rapid prototyping, and developing a mapping to use as a starting point for defining similar mappings.

Open the Code Replacement Tool

Do one of the following:

- In the Command Window, enter the command `crtool`.
- In the Configuration Parameters dialog box, navigate to **All Parameters** > **Code Generation** > **Code replacement library** and click **Custom**.

An Embedded Coder license is not required to create a custom code replacement library. However, you must have an Embedded Coder license to use a such a library.

By default, the tool displays, left to right, a root pane, a list pane, and a dialog pane. You can manipulate the display:

- Drag boundaries to widen, narrow, shorten, or lengthen panes, and to resize table columns.
- Select **View** > **Show dialog pane** to hide or display the right-most pane.
- Click a table column heading to sort the table based on contents of the selected column.
- Right-click a table column heading and select **Hide** to remove the column from the display. (You cannot hide the **Name** column.)

Create a Code Replacement Table

1. In the Code Replacement Tool window, select **File** > **New table**.
2. In the right pane, name the table and click **Apply**. Later, when you save the table, the tool uses the table name that you specify to name the file. For example, if you enter the name `my_sinfcn`, the tool names the file `my_sinfcn.m`.

Create Table Entries

Create one or more table entries. Each entry maps the conceptual representation of a function or operator to your implementation representation. The information that you enter depends on the type of entry you create. Enter the following information:

1. In the left pane, select the table to which you want to add the entry.
2. Select **File** > **New entry** > **entry-type**, where **entry-type** is one of:

   - Math Operation
   - Function
   - BLAS Operation
   - CBLAS Operation
   - Net Slope Fixed-Point Operation
   - Semaphore entry
   - Customization entry

   The new entry appears in the middle pane, initially without a name.
3. In the middle pane, select the new entry.
4. In the right pane, on the **Mapping Information** tab, from the **Function** or **Operation** menu, select the function or operation that you want the code generator to replace. Regardless of the entry type, make a selection from this menu. Your selection determines what other information you specify.

   Except for customization entries, you also specify information for your replacement function prototype. You can also specify implementation attributes, such as the rounding modes to apply.
5. If prompted, specify additional entry information that you want the code generator to use when searching for a match. For example, when you select an addition or subtraction operation, the tool prompts you to specify an algorithm (`Cast before operation` or `Cast after operation`).
6. Review the conceptual argument information that the tool populates for the function or operation. Conceptual input and output arguments represent arguments for the function or operator being replaced. Conceptual arguments observe naming conventions (`'y1'`, `'u1'`, `'u2'`, ...) and data types familiar to the code generator.

   If you do not want the data types for your implementation to be the same as the conceptual argument types, clear the **Make the conceptual and implementation argument types the same** check box. For example, most ANSI-C functions operate on and return `double` data. Clear the check box if want to map a conceptual representation of the function to an implementation representation that specifies an argument and return value. For example, clear the check box to map the conceptual representation of the function `sin` to an implementation representation that

specifies an argument and return value of type `single` (`single sin(single)`), of type `double` (`double sin(double)`. In this case, the code generator produces the following code:

```
y = (single) sin(u1);
```

If you select `Custom` for a function entry, specify only conceptual argument information.

7 Specify the name and argument information for your replacement function. As you enter the information and click **Apply**, the tool updates the **Function signature preview**.

8 Specify additional implementation attributes that apply. For example, depending on the type and name of the entry that you specify, the tool prompts you to specify:

- Integer saturation mode
- Rounding modes
- Whether to allow inputs that include expressions
- Whether a function modifies internal or global state

9 Click **Apply**.

Validate Tables and Entries

The Code Replacement Tool provides a way to validate the syntax of code replacement tables and table entries as you define them. If the tool finds validation errors, you can address them and retry the validation. Repeat the process until the tool does not report errors.

To	Do
Validate table entries	Select an entry, scroll to the bottom of the **Mapping Information** tab, and click **Validate entry**. Alternatively, select one or more entries, right-click, and select **Validate entries**.
Validate a table	Select the table. Then, select **Actions** > **Validate table**.

Save a Table

When you save a table, the tool validates unvalidated content.

1. Select **File** > **Save table**.
2. In the Browse For Folder dialog box, specify a location and name for the file. Typically, you select a location on the MATLAB path. By default, the tool names the file using the name that you specify for the table with the extension `.m`.
3. Click **Save**.

Open and Modify Tables

After saving a code replacement table, to make changes in the table:

1. Select **File** > **Open table**.
2. In the Import file dialog box, browse to the MATLAB file that contains the table.

Repeat the sequence to open and work on multiple tables.

If you open multiple tables, you can manage the tables together. For example, use the tool to:

- Create new table entries.
- Delete entries.
- Copy and paste or cut and paste information between tables.

Define Mappings Programmatically

This example shows how to define a code replacement mapping programmatically. The programming interface for defining code replacement table mappings is ideal for

- Modifying tables that you create with the Code Replacement Tool.
- Defining mappings for specialized entries that you cannot create with the Code Replacement Tool.
- Replicating and modifying similar entries and tables.

Steps for defining a mapping programmatically are:

Create Code Replacement Table

1. Create a table definition file that contains a function definition. For example:

   ```
   function hTable = crl_table_sinfcn()
   ```
2. Within the function body, create the table by calling the function `RTW.TflTable`.

```
hTable = RTW.TflTable;
```

Create Table Entry

For each function or operator that you want the code generator to replace, map a conceptual representation of the function or operator to an implementation representation as a table entry.

1 Within the body of a table definition file, create a code replacement entry object. Call one of the following functions.

Entry Type	Function
Math operation	`RTW.TflCOperationEntry`
Function	`RTW.TflCFunctionEntry`
BLAS operation	`RTW.TflBlasEntryGenerator`
CBLAS operation	`RTW.TflCBlasEntryGenerator`
Fixed-point addition and subtraction operations (support for `SlopesMustBeTheSame` and `ZeroNetBias` parameters)	`RTW.TflCOperationEntryGenerator`
Net slope fixed-point operation	`RTW.TflCOperationEntryGenerator_NetSlope`
Semaphore or mutex entry	`RTW.TflCSemaphoreEntry`
Custom function entry	*`MyCustomFunctionEntry`* (where *`MyCustomFunctionEntry`* is a class derived from `RTW.TflCFunctionEntryML`)
Custom operation entry	*`MyCustomOperationEntry`* (where *`MyCustomOperationEntry`* is a class derived from `RTW.TflCOperationEntryML`)

For example:

```
hEnt = RTW.TflCFunctionEntry;
```

You can combine steps of creating the entry, setting entry parameters, creating conceptual and implementation arguments, and adding the entry to a table with a

single function call to `registerCFunctionEntry`, `registerCPPFunctionEntry`, or `registerCPromotableMacroEntry` if you are creating an entry for a function and the function implementation meets the following criteria:

- Implementation argument names and order match the names and order of corresponding conceptual arguments.
- Input arguments are of the same type.
- The return and input argument names follow the code generator's default naming conventions:
  - Return argument is `y1`.
  - Input arguments are `u1`, `u2`, ..., `un`.

For example:

```
registerCFunctionEntry(hTable, 100, 1, 'sin', 'double', ...
   'sin_dbl', 'double', 'sin_dbl.h','','');
```

As another alternative, you can significantly reduce the amount of code that you write by combining the steps of creating the entry and conceptual and implementation arguments with a call to the `createCRLEntry` function. In this case, specify the conceptual and implementation information as character vector specifications.

For example:

```
hEnt = createCRLEntry(hTable, ...
    'double y1 = sin(double u1)', ...
    'mySin');
```

This approach does not support:

- C++ implementations
- Data alignment
- Operator replacement with net slope arguments
- Entry parameter specifications (for example, priority, algorithm, building information)
- Semaphore and mutex function replacements

Set Entry Parameters

Set entry parameters, such as the priority, algorithm information, and implementation (replacement) function name. Call the function listed in the following table for the entry type that you created.

Entry Type	Function
Math operation	`setTflCOperationEntryParameters`
Function	`setTflCFunctionEntryParameters`
BLAS operation	`setTflCOperationEntryParameters`
CBLAS operation	`setTflCOperationEntryParameters`
Fixed-point addition and subtraction operations where there is a many-to-one mapping, such as a mapping for a range of fixed-point types to the same replacement function (support for `SlopesMustBeTheSame` and `ZeroNetBias` parameters)	`setTflCOperationEntryParameters`
Net slope fixed-point operation	`setTflCOperationEntryParameters`
Semaphore or mutex entry	`setTflCSemaphoreEntryParameters`
Custom function entry	`setTflCFunctionEntryParameters`
Custom operation entry	`setTflCOperationEntryParameters`

To see a list of the parameters that you can set, at the command line, create a new entry and omit the semicolon at the end of the command. For example:

```
hEnt = RTW.TflCFunctionEntry
```

```
hEnt =

  TflCFunctionEntry with properties:

             Implementation: [1x1 RTW.CImplementation]
        SlopesMustBeTheSame: 0
          BiasMustBeTheSame: 0
            AlgorithmParams: []
                   ImplType: 'FCN_IMPL_FUNCT'
      AdditionalHeaderFiles: {0x1 cell}
      AdditionalSourceFiles: {0x1 cell}
```

```
      AdditionalIncludePaths: {0x1 cell}
       AdditionalSourcePaths: {0x1 cell}
          AdditionalLinkObjs: {0x1 cell}
     AdditionalLinkObjsPaths: {0x1 cell}
         AdditionalLinkFlags: {0x1 cell}
      AdditionalCompileFlags: {0x1 cell}
                 SearchPaths: {0x1 cell}
                         Key: ''
                    Priority: 100
               ConceptualArgs: [0x1 handle]
                    EntryInfo: []
                 GenCallback: ''
                 GenFileName: ''
               SaturationMode: 'RTW_SATURATE_UNSPECIFIED'
                RoundingModes: {'RTW_ROUND_UNSPECIFIED'}
          TypeConversionMode: 'RTW_EXPLICIT_CONVERSION'
             AcceptExprInput: 1
                 SideEffects: 0
                  UsageCount: 0
          RecordedUsageCount: 0
                 Description: ''
    StoreFcnReturnInLocalVar: 0
                TraceManager: [1x1 RTW.TflTraceManager]
```

To see the implementation parameters, enter:

```
hEnt.Implemenation
```

```
ans =

  CImplementation with properties:

             HeaderFile: ''
             SourceFile: ''
             HeaderPath: ''
             SourcePath: ''
                 Return: []
          StructFieldMap: []
                   Name: ''
              Arguments: [0x1 handle]
    ArgumentDescriptor: []
```

For example, to set entry parameters for the `sin` function and name your replacement function `sin_dbl`, use the following function call:

```
setTflCFunctionEntryParameters(hEnt, ...
          'Key', 'sin', ...
          'ImplementationName', 'sin_dbl');
```

Create Conceptual Arguments

Create conceptual arguments and add them to the entry's array of conceptual arguments.

- Specify output arguments before input arguments.
- Specify argument names that comply with code generator argument naming conventions:
  - `y1` for a return argument
  - `u1`, `u2`, ..., `un` for input arguments
- Specify data types that are familiar to the code generator.
- The function signature, including argument naming, order, and attributes, must fulfill the signature match sought by function or operator callers.
- The code generator determines the size of the value for an argument with an unsized type, such as integer, based on hardware implementation configuration settings.

For each argument:

1. Identify whether the argument is for input or output, the name, and data type. If you do not know what arguments to specify for a supported function or operation, use the Code Replacement Tool to find them. For example, to find the conceptual arguments for the `sin` function, open the tool, create a table, create a function entry, and in the **Function** menu select `sin`.
2. Create and add the conceptual argument to an entry. You can choose a method from the methods listed in this table.

If	Then

You want simpler code or want to explicitly specify whether the argument is scalar or nonscalar (vector or matrix).	Call the function `createAndAddConceptualArg`. For example:
	`createAndAddConceptualArg(hEnt, ...`
	`  'RTW.TflArgNumeric', ...`
	`  'Name', 'y1',...`
	`  'IOType', 'RTW_IO_OUTPUT',...`
	`  'DataTypeMode', 'double');`

If	Then
	The second argument specifies whether the argument is scalar (`RTW.TflArgNumeric` or`RTW.TflArgMatrix`) .

You want to create an argument based on a built-in argument definition (for example, scalar or nonscalar).	Call `getTflArgFromString` to create the argument. Then, call `addConceptualArg` to add the argument to the entry.
	`arg = getTflArgFromString(hEnt, 'y1','double');`
	`arg.IOType = 'RTW_IO_OUTPUT';`
	`addConceptualArg(hEnt, arg);`

You need to define several similar mappings, you want to minimize the code to write, and the entries do not require data alignment, use net slope arguments, or involve semaphore or mutex replacements.	Call `createCRLEntry` to create the entry and specify conceptual and implementation arguments in a single function call.
	`hEnt = createCRLEntry(hTable, ...`
	`'double y1 = sin(double u1)', ...`
	`'mySin');`

The following code shows the second approach listed in the table for specifying the conceptual output and input argument definitions for the `sin` function.

```
% Conceptual Args

arg = getTflArgFromString(hEnt, 'y1','double');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEnt, arg);

arg = getTflArgFromString(hEnt, 'u1','double');
addConceptualArg(hEnt, arg);
```

Create Implementation Arguments

Create implementation arguments for the C or C++ replacement function and add them to the entry.

- When replacing code, the code generator uses the argument names to determine how it passes data to the implementation function.

- For function replacements, the order of implementation argument names must match the order of the conceptual argument names.
- For operator replacements, the order of implementation argument names do not have to match the order of the conceptual argument names. For example, for an operator replacement for addition, `y1=u1+u2`, the conceptual arguments are `y1`, `u1`, and `u2`, in that order. If the signature of your implementation function is `t myAdd(t u2, t u1)`, where `t` is a valid C type, based on the argument name matches, the code generator passes the value of the first conceptual argument, `u1`, to the second implementation argument of `myAdd`. The code generator passes the value of the second conceptual argument, `u2`, to the first implementation argument of `myAdd`.
- For operator replacements, you can remap operator output arguments to implementation function input arguments.

For each argument:

1. Identify whether the argument is for input or output, the name, and the data type.
2. Create and add the implementation argument to an entry. You can choose a method from the methods listed in this table.

If	Then
You want to populate implementation arguments as copies of previously created matching conceptual arguments	Call the function `copyConceptualArgsToImplementation`. For example: `copyConceptualArgsToImplementation(hEnt);`

You want to create and add implementation arguments individually, or vary argument attributes, while maintaining conceptual argument order	Call functions `createAndSetCImplementationReturn` and`createAndAddImplementationArg` . For example:
	`createAndSetCImplementationReturn(hEnt,`
	`'RTW.TflArgNumeric', ...`
	`'Name', 'y1', ...`
	`'IOType', 'RTW_IO_OUTPUT', ...`
	`'IsSigned', true, ...`
	`'WordLength', 32, ...`
	`'FractionLength', 0);`
	
	`createAndAddImplementationArg(op_entry,`
	`'RTW.TflArgNumeric',...`
	`'Name', 'u1', ...`

If	Then

	`'IOType', 'RTW_IO_INPUT',...`
	`'IsSigned', true,...`
	`'WordLength', 32, ...`
	`'FractionLength', 0 );`

If	Then

You want to minimize the amount of code, or specify constant arguments to pass to the implementation function	Create the argument with a call to the function `getTflArgFromString`. Then, use the convenience method `setReturn` or `addArgument` to specify whether an argument is a return value or argument and to add the argument to the entry's array of implementation arguments. For example:
	
	`arg = getTflArgFromString(hEnt, 'y1','double');`
	`arg.IOType = 'RTW_IO_OUTPUT';`
	`hEnt.Implementation.setReturn(arg);`
	
	`arg = getTflArgFromString(hEnt, 'u1','double');`
	`hEnt.Implementation.addArgument(arg);`
	
	The following call to `getTflArgFromString` passes the constant `0` to argument `u2`:
	
	`arg = getTflArgFromString(hEnt, 'u2', 'int16', 0)`
	`hEnt.Implementation.addArgument(arg);`
	
	For semaphore and mutex entries, use the functions `getTflDWorkFromString` and `addDWorkArg` to create and add a DWork argument to the entry. Then create implementation arguments as shown above with `getTflArgFromString` and the convenience methods `setReturn` and `addArgument`. For example:
	
	`arg = getTflDWorkFromString('d1', 'void*')`
	`hEnt.addDWorkArg(arg);`
	
	`arg = hEnt.getTflArgFromString('y1', 'void');`
	`arg.IOType = 'RTW_IO_OUTPUT';`
	`hEnt.Implementation.setRetrurn(arg);`
	
	`arg = hEnt.getTflArgFromString('u1', 'integer');`
	`hEnt.Implementation.addArgument(arg);`
	
	`arg = hEnt.getTflArgFromString('d1', 'void**');`
	`hEnt.Implementation.addArgument(arg);`

If	Then

You need to define several similar mappings, you want to minimize the code to write, and the entries do not require data alignment, use net slope arguments, or involve semaphore or mutex replacements.	Call `createCRLEntry` to create the entry and specify conceptual and implementation arguments in a single function call.
	`hEnt = createCRLEntry(hTable, ...`
	`'double y1 = sin(double u1)', ...`
	`'mySin');`

The following code shows the third approach listed in the table for specifying the implementation output and input argument definitions for the `sin` function:

```
% Implementation Args

arg = hEnt.getTflArgFromString('y1','double');
arg.IOType = 'RTW_IO_OUTPUT';
hEnt.Implementation.setReturn(arg);

arg = hEnt.getTflArgFromString('u1','double');
hEnt.Implementation.addArgument(arg);
```

Add Entry to Table

Add an entry to a code replacement table by calling the function `addEntry`.

```
addEntry(hTable, hEnt);
```

Validate Entry

After you create or modify a code replacement table entry, validate it by invoking it at the MATLAB command line. For example:

```
hTbl = crl_table_sinfcn

hTbl =

RTW.TflTable
    Version: '1.0'
    AllEntries: [2x1 RTW.TflCFunctionEntry]
    ReservedSymbols: []
```

```
    StringResolutionMap: []
```

If the table includes errors, MATLAB reports them. The following examples shows how MATLAB reports a typo in a data type name:

```
hTbl = crl_table_sinfcn

??? RTW_CORE:tfl:TflTable: Unsupported data type, 'dooble'.

Error in ==> crl_table_sinfcn at 7
hTable.registerCFunctionEntry(100, 1, 'sin', 'dooble', 'sin_dbl', ...
```

Save Table

Save the table definition file. Use the name of the table definition function to name the file, for example, `crl_table_sinfcn.m`.

Next, from your requirements, determine whether you need to specify build information for your replacement code.

More About

- “Code You Can Replace from MATLAB Code” on page 52-5
- “Math Function Code Replacement” on page 52-82
- “Memory Function Code Replacement” on page 52-84
- “Specify In-Place Code Replacement” on page 52-86
- “Replace MATLAB Functions with Custom Code Using coder.replace” on page 52-105
- “Reserved Identifiers and Code Replacement” on page 52-111
- “Customize Match and Replacement Process” on page 52-112
- “Scalar Operator Code Replacement” on page 52-120
- “Addition and Subtraction Operator Code Replacement” on page 52-122
- “Small Matrix Operation to Processor Code Replacement” on page 52-126
- “Matrix Multiplication Operation to MathWorks BLAS Code Replacement” on page 52-130
- “Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement” on page 52-137
- “Remap Operator Output to Function Input” on page 52-143
- “Customize Match and Replacement Process for Operators” on page 52-113

- “Fixed-Point Operator Code Replacement” on page 52-146
- “Binary-Point-Only Scaling Code Replacement” on page 52-154
- “Slope Bias Scaling Code Replacement” on page 52-157
- “Net Slope Scaling Code Replacement” on page 52-160
- “Equal Slope and Zero Net Bias Code Replacement” on page 52-166
- “Data Type Conversions (Casts) and Operator Code Replacement” on page 52-169
- “Shift Left Operations and Code Replacement” on page 52-173
- Replacing Math Functions and Operators
- “Prepare for Code Replacement Library Development” on page 52-29
- “Specify Build Information for Replacement Code” on page 52-47
- “Develop a Code Replacement Library” on page 52-15
- “What Is Code Replacement Customization?” on page 52-3

Specify Build Information for Replacement Code

After you define code replacement mappings, determine whether you need to specify build information for your replacement code. A code replacement table entry can specify build information for the code generator to use when replacing code for a match. For example, specify files for implementation replacement code if you are using a generated makefile and the code generation software compiles the code.

Add build information to an entry:

- Interactively, by using the **Build Information** tab in the Code Replacement Tool.
- Programmatically, by using a MATLAB programming interface.

Build Information

The build information can include:

- Paths and file names for header files
- Paths and file names for source files
- Paths and file names for object files
- Compile flags
- Link flags

Choose an Approach for Specifying Build Information

The following table lists situations to help you decide when to use an interactive or programmatic approach to specifying build information:

Situation	Approach
Creating code replacement entries for the first time.	Code Replacement Tool.
You used the Code Replacement Tool to create the entries for which the build information applies.	Code Replacement Tool to specify the build information quickly .
Rapid prototyping entries.	Code Replacement Tool to generate, register, and test entries quickly.

Situation	Approach
Developing an entry to use as a template or starting point for defining similar entries.	Code Replacement Tool to generate entry code that you can copy and modify.
Modifying existing mappings.	MATLAB Editor to update the programming interface directly.

- If an entry uses header, source, or object files, consider whether to make the files accessible to the code generator. You can copy files to the build folder or you can specify individual file names and paths explicitly.
- If you specify *additional* header files/include paths or source files/paths and you copy files, the compiler and utilities such as `packNGo` might find duplicate instances of files (an instance in the build folder and an instance in the original folder).
- If you choose to copy files to the build folder and you are using the `packNGo` function to relocate static and generated code files to another development environment:
  - In the call to `packNGo`, specify the property-value pair `'minimalHeaders' true` (the default). That setting instructs the function to include the minimal header files required to build the code in the zip file.
  - Do not collocate files that you copy with files that you do not copy. If the `packNGo` function finds multiple instances of the same file, the function returns an error.
- If you use the programming interface, paths that you specify can include tokens. A token is a variable defined as a character vector or cell array of character vectors in the MATLAB workspace that you enclose with dollar signs (`$`*`variable`*`$`). The code generator evaluates and replaces a token with the defined value. For example, consider the path `$myfolder$\folder1`, where `myfolder` is a character vector variable defined in the MATLAB workspace as `'d:\work\source\module1'`. The code generator generates the custom path as `d:\work\source\module1\folder1`.

Specify Build Information Interactively with the Code Replacement Tool

The Code Replacement Tool provides a quick, easy way for you to specify build information for code replacement table entries. It is ideal for getting started with defining a table entry, rapid prototyping, and developing table entries to use as a starting point for defining similar mappings.

1 Determine the information that you must specify.

2. Open the Code Replacement Tool.
3. Select the code replacement table entry for which you want to specify the build information. In the left pane, select the table that contains the entry. In the middle pane, select the entry that you want to modify.
4. In the right pane, select the **Build Information** tab.
5. On the **Build Information** tab, specify your build information.

Parameter	Specify
Implementation header file	File name and extension for the header file the code generator needs to generate the replacement code. For example, `sin_dbl.h`.
Implementation source file	File name and extension for the C or C++ source file the code generator needs to generate the replacement code. For example, `sin_dbl.c`.
Additional header files/include paths	Paths and file names for additional header files the code generator needs to generate the replacement code. For example, `C:\libs\headerFiles` and `C:\libs\headerFiles\common.h`. This parameter adds `-I` to the compile line in the generated makefile.
Additional source files/ paths	Paths and file names for additional source files the code generator needs to generate the replacement code. For example, `C:\libs\srcFiles` and `C:\libs\srcFiles\common.c`. This parameter adds `-I` to the compile line in the generated makefile.
Additional object files/ paths	Paths and file names for additional object files the linker needs to build the replacement code. For example, `C:\libs\objFiles` and `C:\libs\objFiles\common.obj`.
Additional link flags	Flags the linker needs to generate an executable file for the replacement code.
Additional compile flags	Flags the compiler needs to generate object code for the replacement code.
Copy files to build directory	Whether to copy header, source, or object files, which are required to generate replacement

Parameter	Specify
	code, to the build folder before code generation. If you specify files with **Additional header files/include paths** or **Additional source files/ paths** and you copy files, the compiler and utilities such as `packNGo` might find duplicate instances of files.

6 Click **Apply**.

7 Select the **Mapping Information** tab. Scroll to the bottom of that table and click **Validate entry**. The tool validates the changes that you made to the entry.

8 Save the table that includes the entry that you just modified.

Specify Build Information Programmatically

The programming interface for specifying build information for a code replacement entry is ideal for:

- Modifying entries created with the Code Replacement Tool.
- Replicating and then modifying similar entries and tables.

The basic workflow for specifying build information programmatically is:

1 Identify or create the code replacement entry that you want to specify the build information.

2 Determine what information to specify.

3 Specify your build information.

Specify	Action

Implementation header file	Use one of the following:
	• Set properties `ImplementationHeaderFile` and `ImplementationHeaderPath` in a call to `setTflCFunctionEntryParameters`, `setTflCOperationEntryParameters`, or `setTflCSemaphoreEntryParameters`. For example:
	`setTflCFunctionEntryParameters(hEnt, ...`
	`'ImplementationHeaderFile', 'sin_dbl.h', ...`

Specify	Action

	`'ImplementationHeaderPath', 'D:/lib/headerFiles'`
	`'Key', 'sin', ...`
	`'ImplementationName', 'sin_dbl');`
	• Set argument `headerFile` in a call to `registerCFunctionEntry`, `registerCPPFunctionEntry`, or `registerCPromotableMacroEntry`

Implementation source file	Set properties `ImplementationSourceFile` and `ImplementationSourcePath` in a call to `setTflCFunctionEntryParameters`, `setTflCOperationEntryParameters`, or `setTflCSemaphoreEntryParameters`. For example:
	`setTflCFunctionEntryParameters(hEnt, ...`
	`'ImplementationHeaderFile', 'sin_dbl.c', ...`
	`'ImplementationHeaderPath', 'D:/lib/sourceFiles'`
	`'Key', 'sin', ...`
	`'ImplementationName', 'sin_dbl');`

Additional header files/include paths	For each file, specify the file name and path in calls to the functions `addAdditionalHeaderFile` and `addAdditionalIncludePath`. For example:
	`libdir = fullfile('$(MATLAB_ROOT)','..', '..', 'lib');`
	`hEnt = RTW.TflCFunctionEntry;`
	`addAdditionalHeaderFile(hEnt, 'common.h');`
	`addAdditionalIncludePath(hEnt, fullfile(libdir, 'include'));`
	These functions add `-I` to the compile line in the generated makefile.

Specify	Action

Additional source files/paths	For each file, specify the file name and path in calls to the functions `addAdditionalSourceFile` and `addAdditionalSourcePath`. For example:
	`libdir = fullfile('$(MATLAB_ROOT)','..', '..', 'lib');`
	`hEnt = RTW.TflCFunctionEntry;`
	`addAdditionalSourceFile(hEnt, 'common.c');`
	`addAdditionalSourcePath(hEnt, fullfile(libdir, 'src'));`
	These functions add `-I` to the compile line in the generated makefile.

Additional object files/paths	For each file, specify the file name and path in calls to the functions `addAdditionalLinkObj` and `addAdditionalLinkObjPath`. For example:
	`libdir = fullfile('$(MATLAB_ROOT)','..', '..', 'lib');`
	`hEnt = RTW.TflCFunctionEntry;`
	`addAdditionalLinkObj(hEnt, 'sin.o');`
	`addAdditionalLinkObjPath(hEnt, fullfile(libdir, 'bin'));`

Compile flags	Set the entry property `AdditionalCompileFlags` to a cell array of character vectors representing the required compile flags. For example:
	`hEnt = RTW.TflCFunctionEntry;`
	`hEnt.AdditionalCompileFlags = {'-Zi -Wall', '-O3'};`

Link flags	Set the entry property `AdditionalLinkFlags` to a cell array of character vectors representing the required link flags. For example:
	`hEnt = RTW.TflCFunctionEntry;`
	`hEnt.AdditionalCompileFlags = {'-MD -Gy', '-T'};`

Specify	Action

Whether to copy header, source, or object files, which are required to generate replacement code, to the build folder before code generation	Use one of the following:
	• Set property `GenCallback` to `'RTW.copyFileToBuildDir'` in a call to `setTflCFunctionEntryParameters`, `setTflCOperationEntryParameters`, or `setTflCSemaphoreEntryParameters`. For example:
	`setTflCFunctionEntryParameters(hEnt, ...`
	`'ImplementationHeaderFile', 'sin_dbl.h', ...`
	`'ImplementationHeaderPath', 'D:/lib/headerFiles'`
	`'Key', 'sin', ...`
	`'ImplementationName', 'sin_dbl'`
	`'GenCallback', 'RTW.copyFileToBuildDir');`
	• Set argument `genCallback` in a call to `registerCFunctionEntry`, `registerCPPFunctionEntry`, or `registerCPromotableMacroEntry` to `'RTW.copyFileToBuildDir'`.
	If a match occurs for a table entry, a call to the function `RTW.copyFileToBuildDir` copies required files to the build folder.
	If you specify additional header files/include paths or additional source files/paths and you copy files, the compiler and utilities such as `packNGo` might find duplicate instances of files.

4 Save the table that includes the entry that you added or modified.

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 replaced with calls to the optimized function. The optimized function does not reside in the build folder. For the code generator to access the files, copy them into the build folder to be compiled and linked into the application.

The table entry specifies the source and header file names and paths. To request the copy operation, the table entry sets the `genCallback` property to `'RTW.copyFileToBuildDir'` in the call to 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 a match occurs for the table

entry, the function `RTW.copyFileToBuildDir` copies the specified source and header files to the build folder for use during the remainder of the build process.

```
function hTable = make_my_crl_table

hTable = RTW.TflTable;

op_entry = RTW.TflCOperationEntry;
setTflCOperationEntryParameters(op_entry, ...
                'Key',                      'RTW_OP_MUL', ...
                'Priority',                 100, ...
                'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
                'RoundingModes',            {'RTW_ROUND_UNSPECIFIED'}, ...
                'ImplementationName',       's32_mul_s32_s32_sat', ...
                'ImplementationHeaderFile', 's32_mul.h', ...
                'ImplementationSourceFile', 's32_mul.s', ...
                'ImplementationHeaderPath', {fullfile('$(MATLAB_ROOT)','crl')}, ...
                'ImplementationSourcePath', {fullfile('$(MATLAB_ROOT)','crl')}, ...
                'GenCallback',              'RTW.copyFileToBuildDir');
.
.
.
addEntry(hTable, op_entry);
```

The following example uses the functions `addAdditionalHeaderFile`, `addAdditionalIncludePath`, `addAdditionalSourceFile`, `addAdditionalSourcePath`, `addAdditionalLinkObj`, and `addAdditionalLinkObjPath` in addition to the code generation callback function `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', ...
                'RoundingModes',            {'RTW_ROUND_UNSPECIFIED'}, ...
                'ImplementationName',       's32_add_s32_s32', ...
                'ImplementationHeaderFile', 's32_add_s32_s32.h', ...
                'ImplementationSourceFile', 's32_add_s32_s32.c'...
                'GenCallback',              'RTW.copyFileToBuildDir');

addAdditionalHeaderFile(op_entry, 'all_additions.h');
addAdditionalIncludePath(op_entry, fullfile(libdir, 'include'));
addAdditionalSourceFile(op_entry, 'all_additions.c');
addAdditionalSourcePath(op_entry, fullfile(libdir, 'src'));
addAdditionalLinkObj(op_entry, 'addition.o');
addAdditionalLinkObjPath(op_entry, fullfile(libdir, 'bin'));
.
```

```
.
.
addEntry(hTable, op_entry);
```

Next, include your code replacement table in a code replacement library and register the library with the code generator.

More About

- “Define Code Replacement Mappings” on page 52-30
- “Register Code Replacement Mappings” on page 52-56
- “Develop a Code Replacement Library” on page 52-15
- “What Is Code Replacement Customization?” on page 52-3
- “Code You Can Replace from MATLAB Code” on page 52-5

Register Code Replacement Mappings

After you define code replacement entries in a code replacement table, you can include the table in a code replacement library that you register with the code generator. When registered, a library appears in the list of available code replacement libraries that you can choose from when configuring the code generator.

Register a code replacement table as a code replacement library:

- Interactively, by using the Code Replacement Tool
- Programmatically, by using a MATLAB programming interface

Choose an Approach for Creating the Registration File

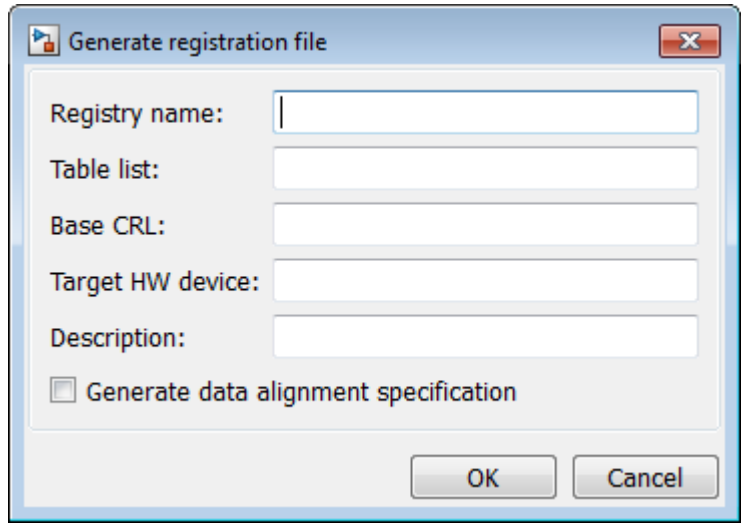
The following table lists situations to help you decide when to use an interactive or programmatic approach to creating a registration file:

If...	Then...
Registering a code replacement table for the first time	Use the Code Replacement Tool.
You used the Code Replacement Tool to create the table	Use the Code Replacement Tool to quickly register the table.
Rapid prototyping code replacement	Use the Code Replacement Tool to quickly generate, register, and test entries.
Creating registration file to use as a template or starting point for defining similar registration files	Use the Code Replacement Tool to generate code that you can copy and modify.
Modifying existing registration files	Use the MATLAB Editor to update the registration file.
Defining multiple code replacement libraries in one registration file	Use the MATLAB Editor to create a new or extend an existing registration file.
Defining code replacement library hierarchy in a registration file	Use the MATLAB Editor to create a new or extend an existing registration file.

Create Registration File Interactively with the Code Replacement Tool

The Code Replacement tool provides a quick, easy way for you to create a registration file for a code replacement table. It is ideal for getting started, rapid prototyping, and generating a registration file that you want to use as a starting point for similar registrations.

1 After you validate and save a code replacement table, select **File > Generate registration file** to open the **Generate registration file** dialog box.



2 Enter the registration information. Minimally, specify:

For...	Specify...
Registry name	Text naming the code replacement library. For example, `Sin Function Example`.
Table list	Text naming one or more code replacement tables to include in the library. Specify each table as one of the following: • Name of a table file on the MATLAB search path • Absolute path to a table file • Path to a table file relative to `$(MATLAB_ROOT)`

For...	Specify...
	You can specify multiple tables. If you do, separate the table specifications with a comma. For example: `crl_table_sinfcn, c:/work_crl/crl_table_muldiv` See "Registration Files That Define Multiple Code Replacement Libraries" on page 52-61 for examples of each type of table specification.

Optionally, you can specify:

For...	Specify...
Description	Text that describes the purpose and content of the library.
Target HW device	Text naming one or more hardware devices the code replacement library supports. Separate names with a comma. To support all device types, enter an asterisk (*). For example, `TI C28x, TI C62x`.
Base CRL	Text naming a code replacement library that you want to serve as a base library for the library you are registering. Use this field to specify library hierarchies. For example, you can specify a general `TI device library` as the base library for a more specific `TI C28x` device library.
Generate data alignment specification	Flag that enables data alignment specification.

Create Registration File Programmatically

The programming interface for creating a registration file for a code replacement table is ideal for:

- Modifying registration files created with the Code Replacement Tool
- Replicating and modifying similar registration files
- Defining multiple code replacement libraries in one registration file

The basic workflow for creating a registration file programmatically consists of the following steps:

1. Define an `rtwTargetInfo` function. The code generator recognizes this function as a customization file. The function definition must include at least the following content:

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@loc_register_crl);

function this = loc_register_crl

this(1) = RTW.TflRegistry;
this(1).Name = 'crl-name';
this(1).TableList = {'table',...};
```

For...	Replace...
`this(1).Name = 'crl-name';`	*`crl-name`* with text naming the code replacement library. For example, `Sin Function Example`.
`this(1).TableList = {'table',...};`	*`table`* with text that identifies the code replacement table that contains your code replacement entries. Specify a table as one of the following: • Name of a table file on the MATLAB search path • Absolute path to a table file • Path to a table file relative to `$(MATLAB_ROOT)` You can specify multiple tables. If you do, separate the table specifications with commas.

Optionally, you can specify:

For...	Replace...
`this(1).Description = '`*`text`*`'`	*`text`* with text that describes the purpose and content of the library.
`this(1).TargetHWDeviceType = {'`*`device-type`*`',...}`	*`device-type`* with text that names a hardware device the code replacement library supports. You can specify multiple device types. Separate device types with a comma. For example, `TI C28x, TI C62x`. To support all device types, enter an asterisk (*).
`this(1).BaseTfl = '`*`base-lib`*`'`	*`base-lib`* with text that names a code replacement library that you want to serve as a base library for the library you are registering. Use this field to specify library hierarchies. For example, you can specify a general `TI device` library as the base library for a `TI C28x device` library. See "Registration Files That Define Code Replacement Library Hierarchies" on page 52-61 for an example.

For example:

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@loc_register_crl);

function this = loc_register_crl

this(1) = RTW.TflRegistry;
this(1).Name = 'Sin Function Example';
this(1).TableList = {'crl_table_sinfcn'};
this(1).TargetHWDeviceType = {'*'};
this(1).Description = 'Example - sin function replacement';
```

2 Save the file with the name `rtwTargetInfo.m`.

3 Place the file on the MATLAB path. When the file is on the MATLAB path, the code generator reads the file after starting and applies the customizations during the current MATLAB session.

Register a Code Replacement Library

Before you can use the code replacement tables defined in a registration file, you must refresh Simulink customizations within the current MATLAB session. To initiate a refresh, enter the following command:

```
sl_refresh_customizations
```

Registration Files That Define Multiple Code Replacement Libraries

Use the programming interface to create a registration file that defines a code replacement library that includes multiple code replacement tables. The following example defines a library that includes multiple tables. The `TableList` fields specify tables that reside at different locations. The tables reside on the MATLAB search path or at locations specified with a path.

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@locCrlRegFcn);

function thisCrl = locCrlRegFcn

  % Register a code replacement library for use with model: rtwdemo_crladdsub
  thisCrl(1) = RTW.TflRegistry;
  thisCrl(1).Name = 'Addition & Subtraction Examples';
  thisCrl(1).Description = 'Example of addition/subtraction op replacement';
  thisCrl(1).TableList = {'crl_table_addsub'};
  thisCrl(1).TargetHWDeviceType = {'*'};

  % Register a code replacement library for use with model: rtwdemo_crlmuldiv
  thisCrl(2) = RTW.TflRegistry;
  thisCrl(2).Name = 'Multiplication & Division Examples';
  thisCrl(2).Description = 'Example of mult/div op repl for built-in integers';
  thisCrl(2).TableList = {'c:/work_crl/crl_table_muldiv'};
  thisCrl(2).TargetHWDeviceType = {'*'};

  % Register a code replacement library for use with model: rtwdemo_crlfixpt
  thisCrl(3) = RTW.TflRegistry;
  thisCrl(3).Name = 'Fixed-Point Examples';
  thisCrl(3).Description = 'Example of fixed-point operator replacement';
  thisCrl(3).TableList = {fullfile('$(MATLAB_ROOT)', ...
     'toolbox','rtw','rtwdemos','crl_demo','crl_table_fixpt')};
  thisCrl(3).TargetHWDeviceType = {'*'};
```

Registration Files That Define Code Replacement Library Hierarchies

Using the programming interface, you can organize multiple code replacement libraries in a hierarchy. The following example shows a registration file that defines four code

replacement tables organized in a hierarchy of four code replacement libraries. The tables include entries that increase in specificity: common entries, entries for TI devices, entries for TI C6xx devices, and entries specific to the TI C67x device.

```
function rtwTargetInfo(cm)

cm.registerTargetInfo(@locCrlRegFcn);

function thisCrl = locCrlRegFcn

  % Register a code replacement library that includes common entries
  thisCrl(1) = RTW.TflRegistry;
  thisCrl(1).Name = 'Common Replacements';
  thisCrl(1).Description = 'Common code replacement entries shared by other libraries';
  thisCrl(1).TableList = {'crl_table_general'};
  thisCrl(1).TargetHWDeviceType = {'*'};

  % Register a code replacement library for TI devices
  thisCrl(2) = RTW.TflRegistry;
  thisCrl(2).Name = 'TI Device Replacements';
  thisCrl(2).Description = 'Code replacement entries shared across TI devices';
  thisCrl(2).TableList = {'crl_table_TI_devices'};
  thisCrl(2).TargetHWDeviceType = {'TI C28x', 'TI C55x', 'TI C62x', 'TI C64x', 'TI 67x'};
  thisCrl(1).BaseTfl = 'Common Replacements'

  % Register a code replacement library for TI c6xx devices
  thisCrl(3) = RTW.TflRegistry;
  thisCrl(3).Name = 'TI c6xx Device Replacements';
  thisCrl(3).Description = 'Code replacement entries shared across TI C6xx devices';
  thisCrl(3).TableList = {'crl_table_TIC6xx_devices'};
  thisCrl(3).TargetHWDeviceType = {'TI C62x', 'TI C64x', 'TI 67x'};

  % Register a code replacement library for the TI c67x device
  thisCrl(3) = RTW.TflRegistry;
  thisCrl(3).Name = 'TI c67x Device Replacements';
  thisCrl(3).Description = 'Code replacement entries for the TI C67x device';
  thisCrl(3).TableList = {'crl_table_TIC67x_device'};
  thisCrl(3).TargetHWDeviceType = {'TI 67x'};
```

After registering your code replacement mappings, verify that code replacements occur.

More About

- “Troubleshoot Code Replacement Library Registration” on page 52-63
- “Specify Build Information for Replacement Code” on page 52-47
- “Verify Code Replacements” on page 52-64
- “Develop a Code Replacement Library” on page 52-15
- “What Is Code Replacement Customization?” on page 52-3
- “Code You Can Replace from MATLAB Code” on page 52-5

Troubleshoot Code Replacement Library Registration

If a code replacement library is not listed as a configuration option or does not appear in the Code Replacement Viewer:

- Refresh the library registration information within the current MATLAB session (`RTW.TargetRegistry.getInstance('reset');` or for the Simulink environment,`sl_refresh_customizations`).
- See whether the registration file, `rtwTargetInfo.m`, contains an error.

More About

- "Register Code Replacement Mappings" on page 52-56

Verify Code Replacements

After you create or modify a code replacement table, use the following techniques to examine and validate the table and its entries.

- Invoke the table definition file at the command prompt.
- Use the Code Replacement Viewer to examine libraries, tables, and entries.
- Trace code replacements from the source where you applied the code replacement library.
- Examine code replacement hits and misses logged during code generation.

Code Replacement Hits and Misses

The code generator logs code replacement table entries for which it finds and does not find matches in the hit cache and miss cache, respectively. When a code replacement entry match fails and code is not replaced, the code generator logs the call site object (CSO) for the miss in the miss cache. When an entry match succeeds, the code generator logs the matched entry in the hit cache.

The code generator overwrites the hit and miss cache data each time it produces code. The cache data reflects hits and misses for only the last application component (MATLAB code or Simulink model) for which you generate code.

You can use the Code Replacement Viewer to review trace information based on logged hit and miss trace data. The hit cache provides trace information that helps to verify code replacements.

The miss cache and related miss data collected and stored in code replacement tables provide trace information for misses. Use this information for misses to troubleshoot expected code replacements that do not occur. Trace information for a miss:

- Identifies the call site object.
- Provides a link to the relevant source location for the miss.
- Includes information about the reason for the miss.

Validate a Table Definition File

After you create or modify a code replacement table definition file, validate it. At the command prompt, specify the name of the table in a call to the `isvalid` function. For example:

```
isvalid(crl_table_sinfcn)

ans =

      1
```

MATLAB displays errors that occur. In the following example, MATLAB detects a typo in a data type name.

```
isvalid(crl_table_sinfcn)

??? RTW_CORE:tfl:TflTable: Unsupported data type, 'dooble'.

Error in ==> crl_table_sinfcn at 7
hTable.registerCFunctionEntry(100, 1, 'sin', 'dooble', 'sin_dbl', ...
```

Review Library Content

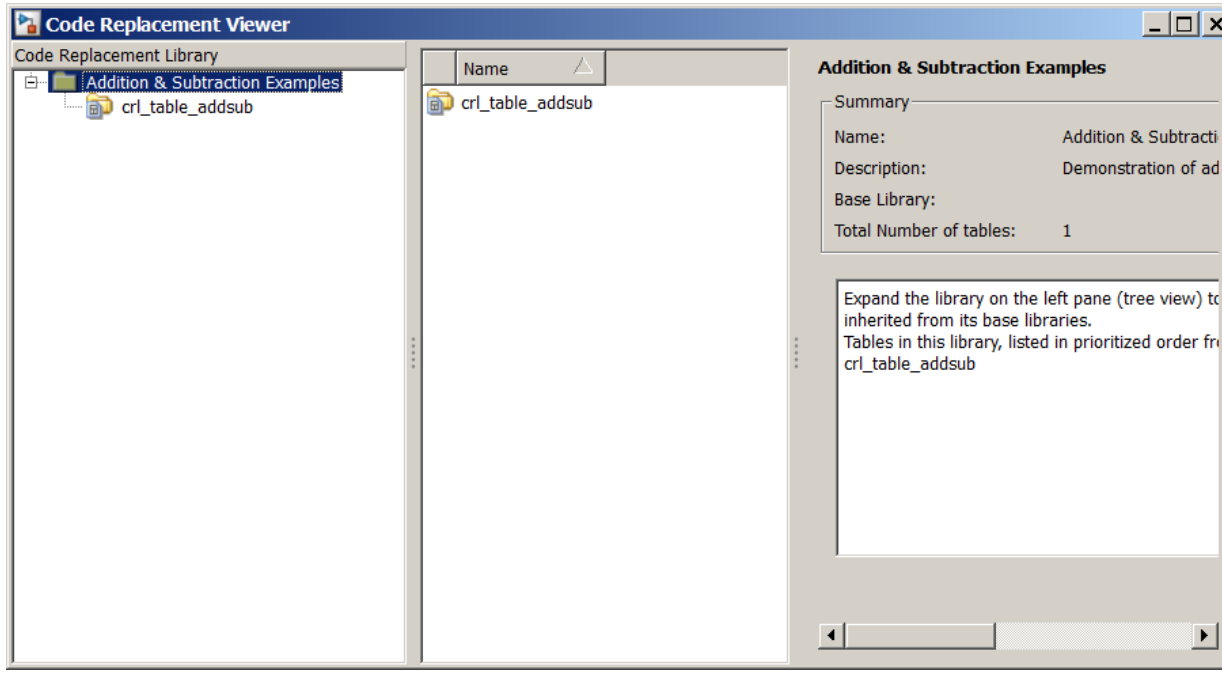
After you create or modify a code replacement library, use the Code Replacement Viewer to review and verify the list of tables in the library and the entries in each table.

1 Open the viewer to display the contents of your library. At the command prompt, enter the following command:

```
crviewer('library')
```

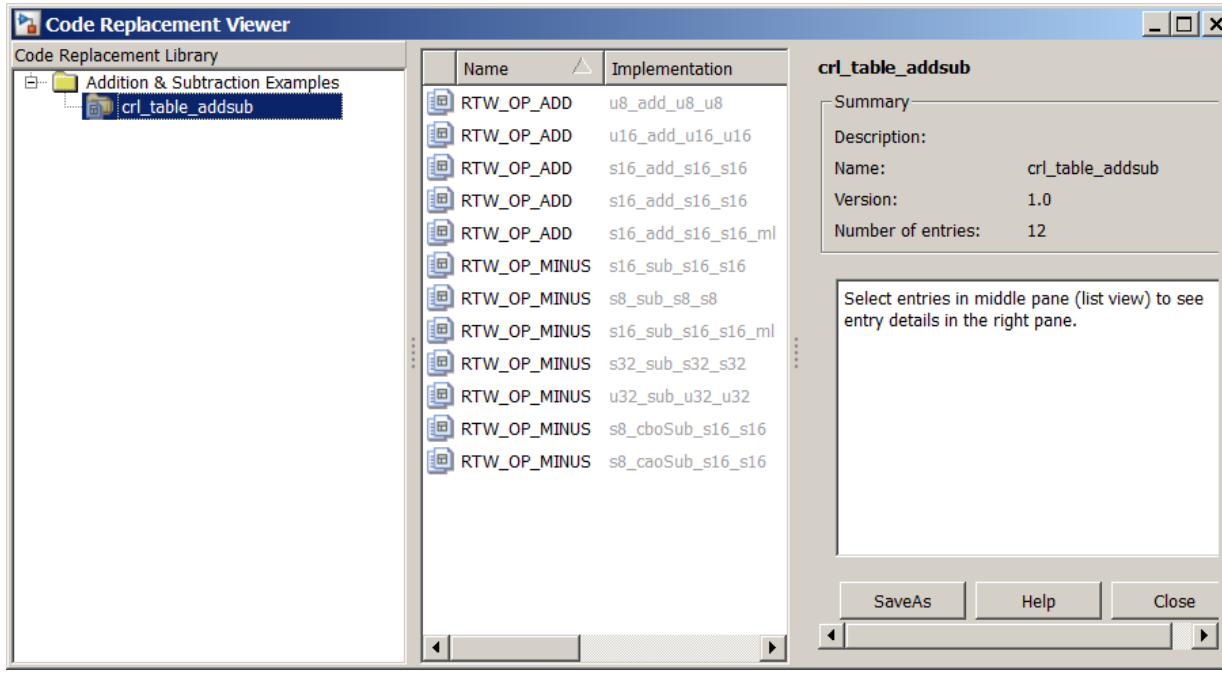
For example:

```
crviewer('Addition & Subtraction Examples')
```



2 Review the list of tables in the left pane. Are tables missing? Are the tables listed in the correct relative order? By default, the viewer displays tables in search order.

3 In the left pane, click each table and review the list of entries in the center pane. Are entries missing? Does the list include extraneous or unexpected entries?



Review Table Content

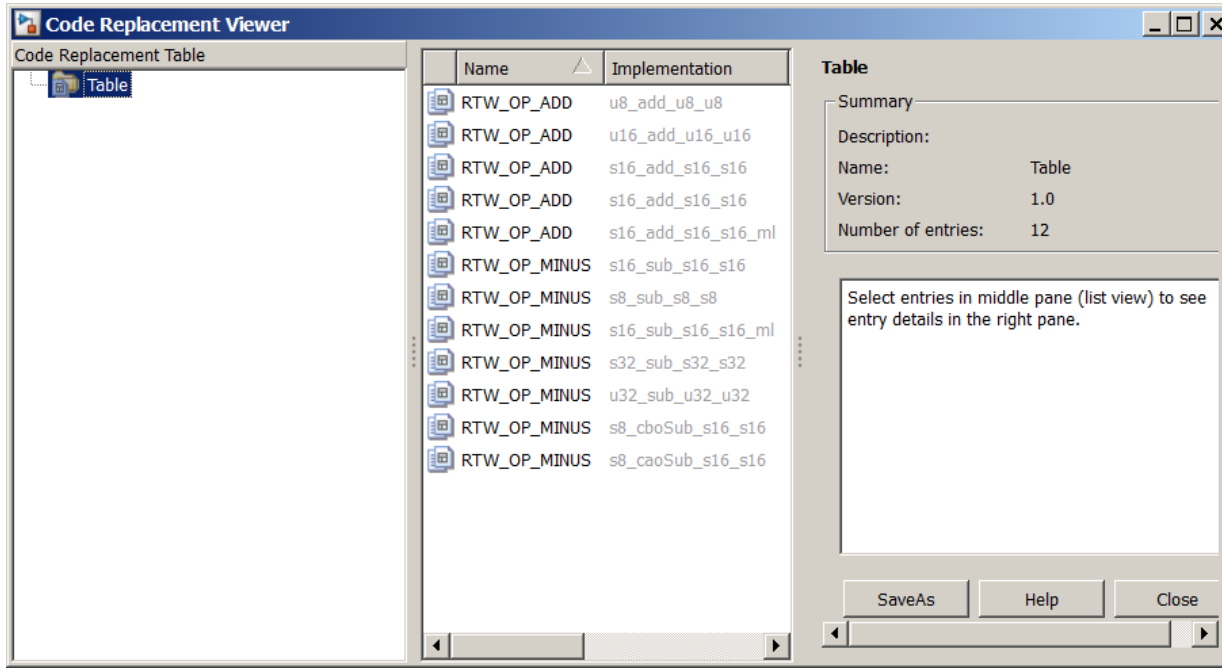
After you create or modify a code replacement table, use the Code Replacement Viewer to review and verify table entries.

1 Open the viewer to display the contents of your table. At the command prompt, enter the following command. *`table`* is a MATLAB file that defines code replacement tables. The file must be in the current folder or on the MATLAB path.

```
crviewer(table)
```

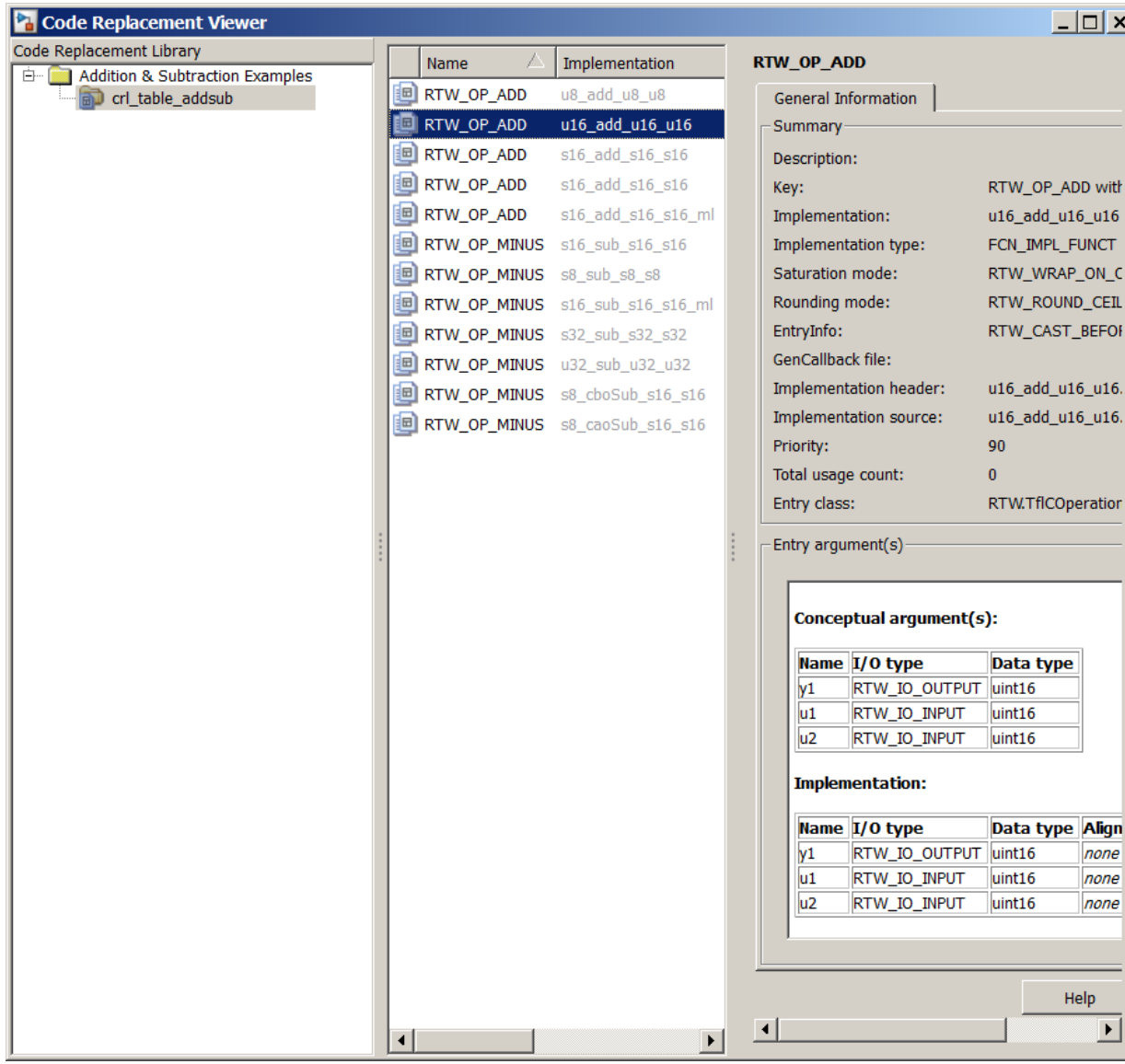
For example:

```
crviewer(crl_table_addsub)
```



2 Review the list of entries in the center pane. Are entries missing? Does the list include extraneous or unexpected entries? By default, the viewer displays entries in search order.

3 In the center pane, click each entry and verify the entry information in the right pane.



- Argument order is correct.

- Conceptual argument names match code generator naming conventions.
- Implementation argument names are correct.
- Algorithm properties (for example, saturation and rounding mode) are set correctly.
- Header or source file specification is not missing.
- I/O types are correct.
- Relative priority of entries is correct.

Review Code Replacements

After you review the content of your code replacement library and tables, generate code and a code generation report. Verify that the code generator replaces code as you expect.

The Code Replacements Report details the code replacement library functions that the code generator uses for code replacements. The report provides a mapping between each replacement instance and the line of MATLAB code that triggered the replacement. The Code Replacements report is not available for generated MEX functions.

The following example illustrates two complementary approaches for reviewing code replacements:

- Check the Code Replacements Report section of the code generation report for expected replacements.
- Trace code replacements.

1. Identify the MATLAB function where you anticipate that a function or operator replacement occurs. This example uses the function matlabroot/toolbox/rtw/rtwdemos/crl_demo/addsub_two_int16.m.

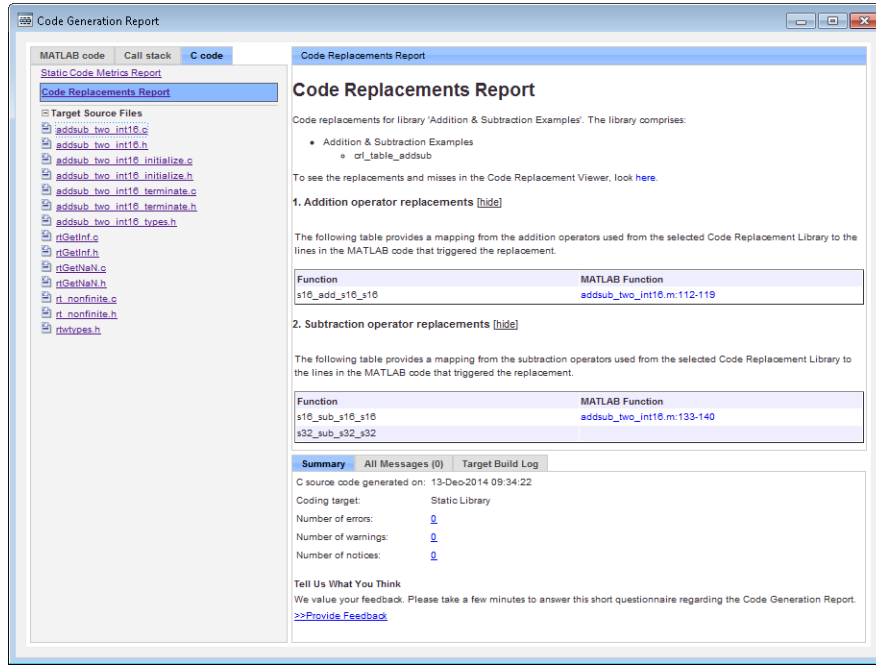
   ```
   function [y1, y2] = addsub_two_int16(u1, u2)

   y1 = int16(u1 + u2);
   y2 = int16(u1 - u2);
   ```

2. Identify or create code or a script to exercise the function. For example, consider test file `addsub_to_int16_test.m`, which includes the following code:

   ```
   disp('Input')
   u1 = int16(10)
   ```

```
u2 = int16(10)

[y1, y2] = addsub_two_int16(u1, u2);

disp('Output')
disp('y1 =')
disp(y1);
disp('y2 =')
disp(y2);
```

3. Open the MATLAB Coder app.
4. On the **Select Source Files** page, add your function to the project. For this example, add function `addsub_two_int16`. Click **Next**.
5. On the **Define Input Types** page, use the test file `addsub_to_int16_test` to automatically define the input types. Click **Next**.
6. On the **Check for Run-Time Issues** page, specify the test file `addsub_to_int16_test`. The app runs the test file, replacing calls to `addsub_to_int16_test` with calls to a MEX version of `addsub_to_int16_test`. Click **Next**.
7. To open the **Generate** dialog box, on the **Generate Code** page, click the **Generate** arrow  .
8. Set **Build type** to generate source code. Before you build an executable, you want to review your code replacements in the generated code.
9. In the Generate dialog box, click **More Settings**.
10. Configure the code generator to use your code replacement library. On the **Custom Code** tab, set the **Code replacement library** parameter to the name of your library. For this example, set the library to `Addition & Subtraction Examples`.
11. Configure the code generation report to include the Code Replacements Report. On the **Debugging** tab, select:
    - **Always create a code generation report**
    - **Code replacements**
    - **Automatically launch a report if one is generated**
12. To generate code and a report, click **Generate**.
13. Open the **Code Replacements Report** section of the code generation report.



That report lists the replacement functions that the code generator used. The report provides a mapping between each replacement instance and the MATLAB code that triggered the replacement.

Review the report:

- Check whether expected function and operator code replacements occurred.
- In the replacements sections, click each code link to see the source that triggered the reported code replacement.

If a function or operator is not replaced as expected, the code generator used a higher-priority (lower-priority value) match or did not find a match.

To analyze and troubleshoot code replacement misses, use the trace information that the Code Replacement Viewer provides. See “Troubleshoot Code Replacement Misses” on page 52-74.

More About

- "Troubleshoot Code Replacement Misses" on page 52-74
- "Register Code Replacement Mappings" on page 52-56
- "Deploy Code Replacement Library" on page 52-81
- "Develop a Code Replacement Library" on page 52-15
- "What Is Code Replacement Customization?" on page 52-3

Troubleshoot Code Replacement Misses

Use miss reason messages that appear in the Code Replacement Viewer to analyze and correct code replacement misses.

Miss Reason Messages

The Code Replacement Viewer displays miss reason messages in trace information for code replacement misses. A legend listing each message that appears in the miss report precedes the report details. A message consists of:

- Numeric identifier, which identifies the message in the report details.
- Message text, which in some cases includes placeholders for names of arguments, call site object values, table entry values, and property names.

For example:

```
1. Mismatched data types (argument name, CSO value, table entry value)
```

The parenthetical information represents placeholders for actual values that appear in the report details.

In the **Miss Source Locations** table that lists the miss details, the **Reason** column includes:

- The message identifier, as listed in the legend.
- The placeholder values for that instance of the miss reason message.

The following **Reason** details indicate a data type mismatch because the call site object specifies data type `int8` for arguments `y1`, `u1`, and `u2`, while the code replacement table entry specifies `uint32`.

```
1. y1, int8, uint32
   u1, int8, uint32
   u2, int8, uint32
```

Depending on your situation and the reported miss reason, troubleshoot reported misses by looking for instances of the following:

- A typo in the code replacement table entry definition or a source parameter setting.
- Information missing from the code replacement table entry or a source parameter setting.

- Invalid or incorrect information in the code replacement table entry definition or a source parameter setting.
- Arguments incorrectly ordered in the code replacement table entry definition or the source being replaced with replacement code.
- Failed algorithm classification for an addition or subtraction operation due to:
  - An ideal accumulator not being calculated because the type of an input argument is not fixed-point or the slope adjustment factors of the input arguments are not equal.
  - Input or output casts with a floating-point cast type.
  - Input or output casts with cast types that have different slope adjustment factors or biases.
  - Output casts not being convertible to a single output cast.
  - Input casts resulting in loss of bits.

Analyze and Correct Code Replacement Misses

The following example shows how to use Code Replacement Viewer trace information to troubleshoot code replacement misses. You must have already reviewed and tested code replacements for your MATLAB code.

1 Review the code generated for a specific code element, looking for expected code replacement. Regenerate or reopen the code generation report for your MATLAB code. If you already generated the code generation report that includes the Code Replacements Report for matlabroot/toolbox/rtw/rtwdemos/crl_demo/addsub_two_int16.m, open the file `codegen/lib/addsub_two_int16/html/index.html`. For information on how to regenerate the report, see "Verify Code Replacements" on page 52-64.

   To examine the code generated for function, from the code generation report, open the generated file `addsub_two_int16.c`.

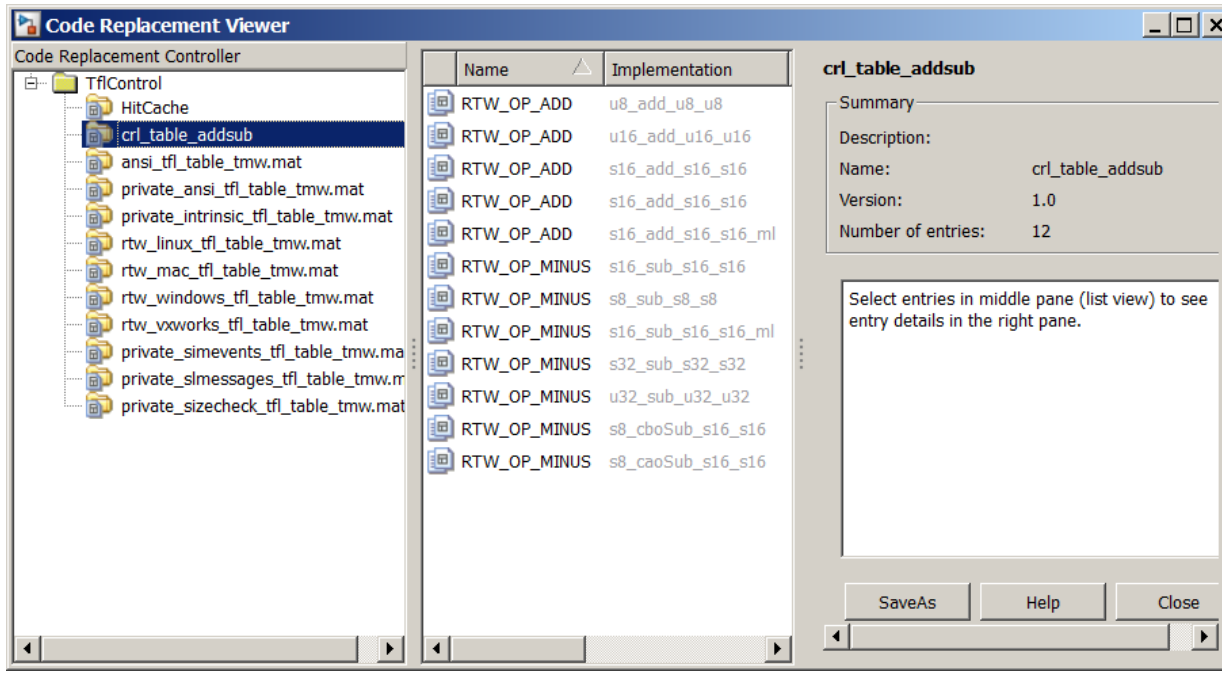
```
21    */
22   void addsub_two_int16(short u1, short u2, short *b_y1, short *y2)
23   {
24     *b_y1 = s16_add_s16_s16(u1, u2);
25     *y2 = s16_sub_s16_s16(u1, u2);
26   }
```

The code generator replaced code, but the replacement is for the signed version of the 16-bit addition and subtraction operations. You expected code replacements for operations on unsigned data.

2 Open the Code Replacements Report for the MATLAB code.

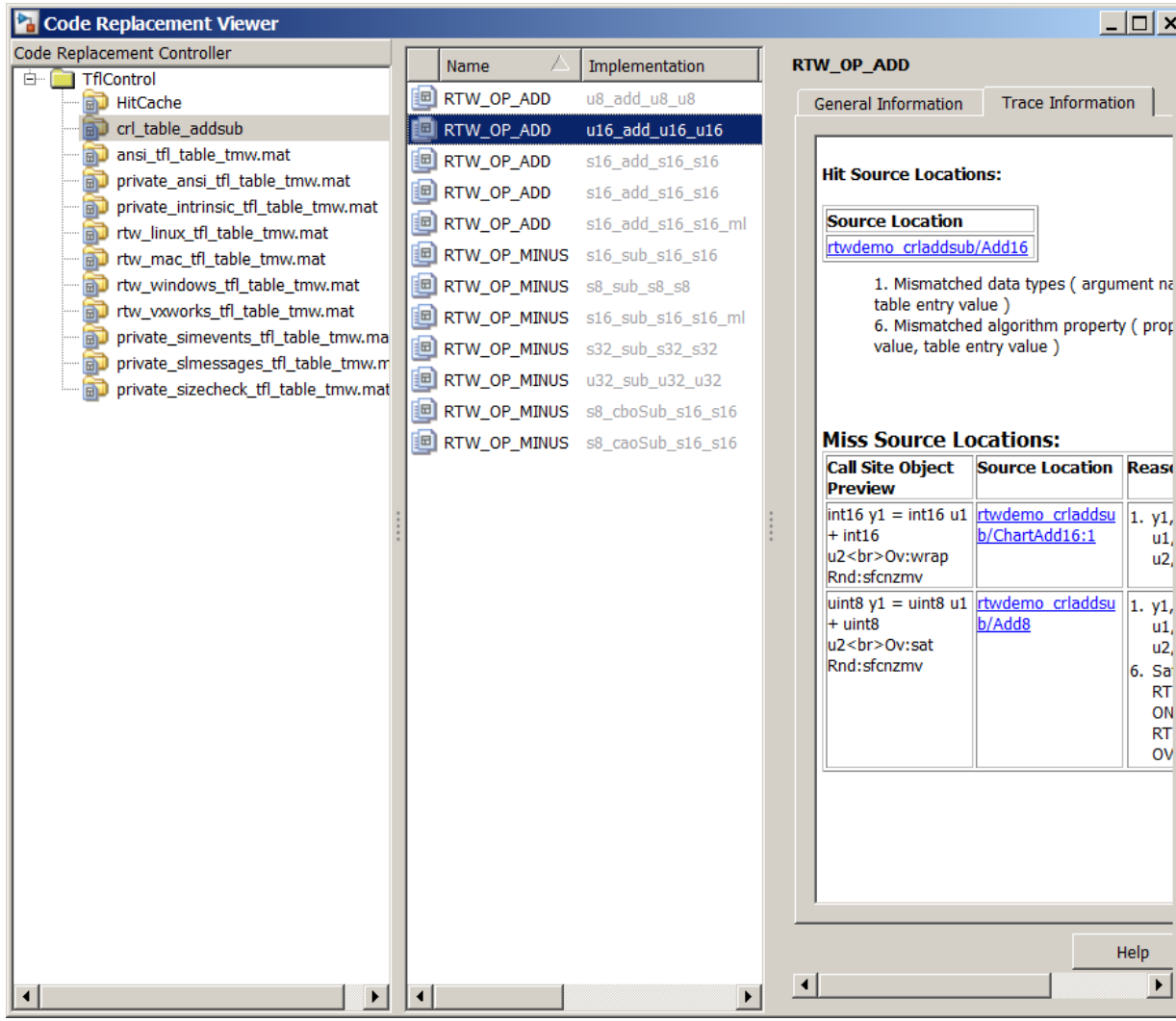
3 Click the link to open the Code Replacement Viewer.

4 In the viewer left pane, select your code replacement table. The following display shows entries for code replacement table `crl_table_addsub`.



5 In the middle pane, select table entry `RTW_OP_ADD` with implementation function `u16_add_u16_u16`.

6 In the right pane, select the **Trace Information** tab.



The **Trace Information** is a table that lists the following information for each miss:

- Call site object preview. The call site object is the conceptual representation of addition operator. The code generator uses this object to query the code replacement library for a match.

- A link to the source location in the MATLAB function where the code generator considered replacing code.
- The reasons that the miss occurred. See “Miss Reason Messages” on page 52-74.

For this example, the report shows misses for function `addsub_two_int16.m`.

7 Find that source in the trace information. Depending on your situation and the reported miss reason, consider looking for a condition such as a typo in the code replacement table entry definition or a source parameter setting. For a list of conditions to consider, see“Miss Reason Messages” on page 52-74.

For this example, determine why code for function `addsub_two_int16` is not replaced with code for an unsigned 16-bit addition operation. The miss reasons for the function indicate data type and algorithm mismatches. For the three arguments:

- The data type in the call site object is a signed 16-bit integer. The code replacement entry specifies an unsigned 16-bit integer.
- The algorithm property in the call site object is `RTW_SATURATE_ON_OVERFLOW` while the code replacement entry specifies `RTW_WRAP_ON_OVERFLOW`.

8 Correct the specified MATLAB code and relevant specifications or code replacement table entry. If the issue concerns the MATLAB code, use the source location in the trace information to find the code to correct. For this example, you expected an unsigned addition operation to occur for the `addsub_two_int16` function.

To fix the mismatches, in the test file `addsub_to_int16_test`, change the data types definitions for `u1` and `u2` as follows:

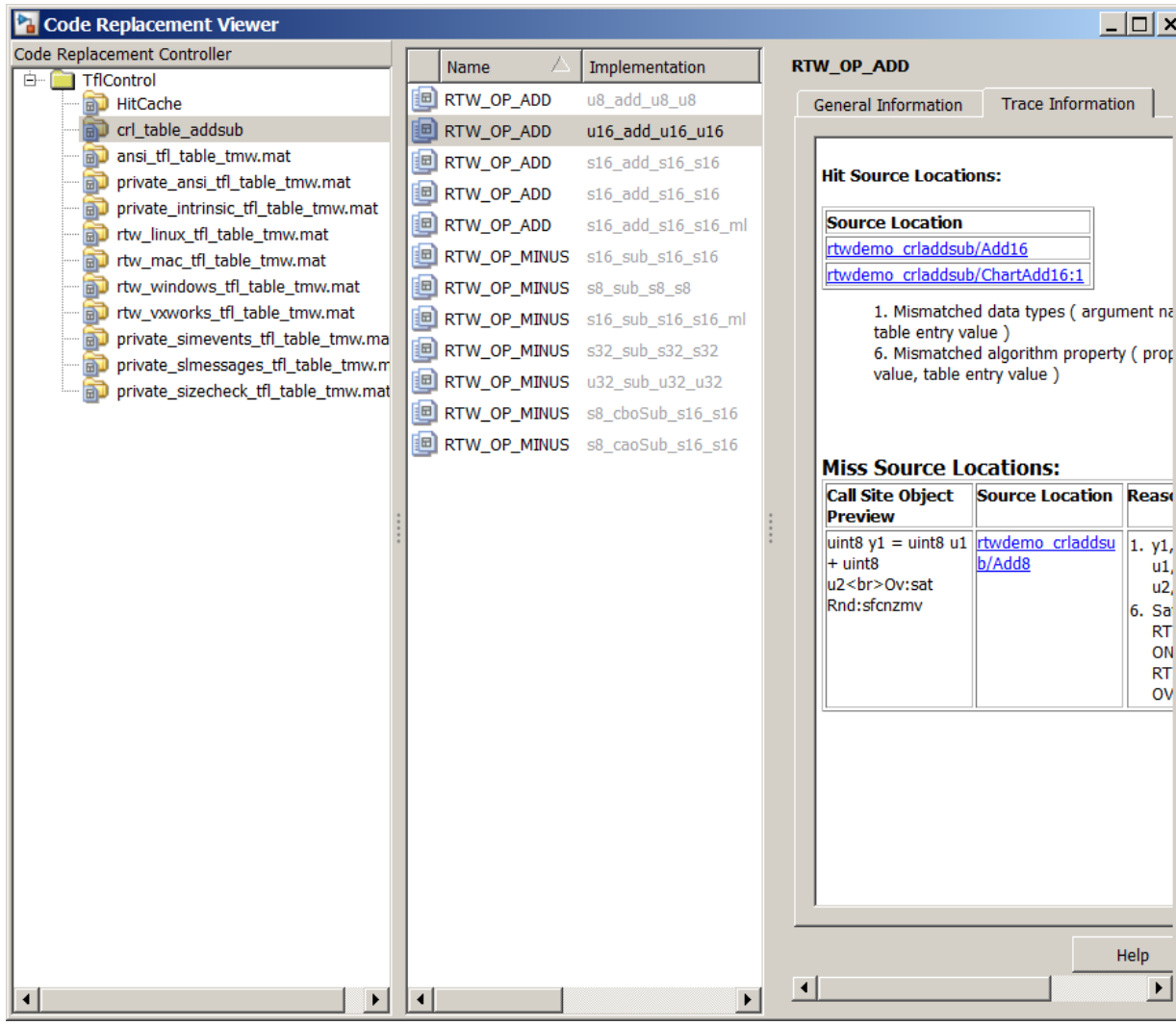
```
u1 = uint16(10)
u2 = uint16(10)
```

In the MATLAB Coder app:

- Open the project that contains the `addsub_to_int16` function.
- Use the updated test file `addsub_to_int16_test` to automatically redefine the input types.
- Run the test file.
- In the project settings dialog box, on the **Speed** tab, clear the check box for the **Saturate on integer overflow** parameter.

- Regenerate code and a report.

9 From the Code Replacements Report, open the Code Replacement Viewer. Use the Code Replacement Viewer trace information to verify that your MATLAB code or code replacement table entry corrects the code replacement issue. In the following display, the trace information shows a hit for function `addsub_two_int16`.



More About

- "Verify Code Replacements" on page 52-64

Deploy Code Replacement Library

After you verify code replacements and are ready to package and deploy a code replacement library for others to use:

1. Move your code replacement table files to an area that is on the MATLAB search path and that is accessible to and shared by other users.
2. Move the `rtwTargetInfo.m` registration file, to an area that is on the MATLAB search path and that is accessible to and shared by other users. If you are deploying a library to a folder in a development environment that already contains a `rtwTargetInfo.m` file, copy the registration code from your code replacement library version of `rtwTargetInfo.m` and paste it into the shared version of that file.
3. Register the library customizations or restart MATLAB.
4. Verify that the libraries are available for configuring the code generator and that code replacements occur as expected.
5. Inform users that the libraries are available and provide direction on when and how to apply them.

More About

- "Verify Code Replacements" on page 52-64
- "Package Code for Other Development Environments" (MATLAB Coder)
- "Develop a Code Replacement Library" on page 52-15
- "What Is Code Replacement Customization?" on page 52-3

Math Function Code Replacement

This example shows how to define a code replacement mapping for a math function. The example defines a mapping for the `sin` function programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

1. Create a table definition file that contains a function definition. For example:

```
function hTable = crl_table_sinfcn2()
%CRL_TABLE_SINFCN2 - Define function entry for code replacement table.
```

2. Within the function body, create the table by calling the function `RTW.TflTable`.

```
hTable = RTW.TflTable;
```

3. Create an entry for the function mapping with a call to the `RTW.TflCFunctionEntry` function.

```
% Create entry for sin function replacement
fcn_entry = RTW.TflCFunctionEntry;
```

4. Set function entry parameters with a call to the `setTflCFunctionEntryParameters` function.

```
setTflCFunctionEntryParameters(fcn_entry, ...
                               'Key',                    'sin', ...
                               'Priority',               30, ...
                               'ImplementationName',     'mySin', ...
                               'ImplementationHeaderFile', 'basicMath.h',...
                               'ImplementationSourceFile', 'basicMath.c');
```

5. Create conceptual arguments `y1` and `u1`. There are multiple ways to set up the conceptual arguments. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call.

```
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');
```

6. Copy the conceptual arguments to the implementation arguments. There are multiple ways to set up the implementation arguments. This example uses a call to the `copyConceptualArgsToImplementation` function to create and add implementation arguments to the entry by copying matching conceptual arguments.

```
copyConceptualArgsToImplementation(fcn_entry);
```

7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, fcn_entry);
```

8 Save the table definition file. Use the name of the table definition function to name the file.

More About

- "Define Code Replacement Mappings" on page 52-30
- "Specify In-Place Code Replacement" on page 52-86
- "Reserved Identifiers and Code Replacement" on page 52-111
- "Customize Match and Replacement Process" on page 52-112
- "Develop a Code Replacement Library" on page 52-15

Memory Function Code Replacement

This example shows how to define a code replacement mapping for a memory function. The example defines a mapping for the `memcpy` function programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

1. Create a table definition file that contains a function definition. For example:

   ```
   function hTable = crl_table_memcpy()
   ```

2. Within the function body, create the table by calling the function `RTW.TflTable`.

   ```
   hTable = RTW.TflTable;
   ```

3. Create an entry for the function mapping with a call to the `RTW.TflCFunctionEntry` function.

   ```
   % Create entry for void* memcpy(void*, void*, size_t)
   fcn_entry = RTW.TflCFunctionEntry;
   ```

4. Set function entry parameters with a call to the `setTflCFunctionEntryParameters` function.

   ```
   % Set SideEffects to 'true' for function returning void to prevent it from
   % being optimized away.
   setTflCFunctionEntryParameters(fcn_entry, ...
                                  'Key',                    'memcpy', ...
                                  'Priority',               90, ...
                                  'ImplementationName',     'memcpy_int', ...
                                  'ImplementationHeaderFile', 'memcpy_int.h',...
                                  'SideEffects',            true);
   ```

5. Create conceptual arguments `y1`, `u1`, `u2`, and `u3`. There are multiple ways to set up the conceptual arguments. This example uses calls to the `getTflArgFromString` and `addConceptualArg` functions to create and add the arguments.

   ```
   arg = getTflArgFromString(hTable, 'y1', 'void*');
   arg.IOType = 'RTW_IO_OUTPUT';
   addConceptualArg(fcn_entry, arg);

   arg = getTflArgFromString(hTable, 'u1', 'void*');
   addConceptualArg(fcn_entry, arg);

   arg = getTflArgFromString(hTable, 'u2', 'void*');
   addConceptualArg(fcn_entry, arg);

   arg = getTflArgFromString(hTable, 'u3', 'size_t');
   addConceptualArg(fcn_entry, arg);
   ```

6. Copy the conceptual arguments to the implementation arguments. There are multiple ways to set up the implementation arguments. This example uses a call

to the `copyConceptualArgsToImplementation` function to create and add implementation arguments to the entry by copying matching conceptual arguments.

```
copyConceptualArgsToImplementation(fcn_entry);
```

7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, fcn_entry);
```

8 Save the table definition file. Use the name of the table definition function to name the file.

More About

- "Code You Can Replace from MATLAB Code" on page 52-5
- "Define Code Replacement Mappings" on page 52-30
- "Specify In-Place Code Replacement" on page 52-86
- "Reserved Identifiers and Code Replacement" on page 52-111
- "Customize Match and Replacement Process" on page 52-112
- "Develop a Code Replacement Library" on page 52-15

Specify In-Place Code Replacement

In-place code replacement is an optimization technique that uses a single buffer, that is, the same memory, to store function input and output data, as in `x=foo(x)`.

When you generate C or C++ code from MATLAB code, the code generator supports in-place function argument code replacement. When you interactively create a code replacement table entry with the Code Replacement Tool, you can specify in-place function argument replacement. You can also specify in-place function argument replacement programmatically with the Code Replacement Library API.

Argument Specification Requirements

- The argument must be a pointer.
- An argument can be in-place with only one other argument.
- Specify an input argument as in-place (shares memory) with an output argument or an output argument as in-place with an input argument.

Interactive Argument Replacement Specification with Code Replacement Tool

This example shows how to specify in-place function argument replacement when replacing code for a MATLAB function with the Code Replacement Tool. The tool enforces in-place argument specification requirements as you add arguments and modify argument properties.

1. Create the following MATLAB function, `customFunction.m`.

   ```
   function x = customFunction(x)
   % Function that updates the input and returns it as an output

   coder.replace('-errorifnoreplacement');
   x = sin(x);
   ```

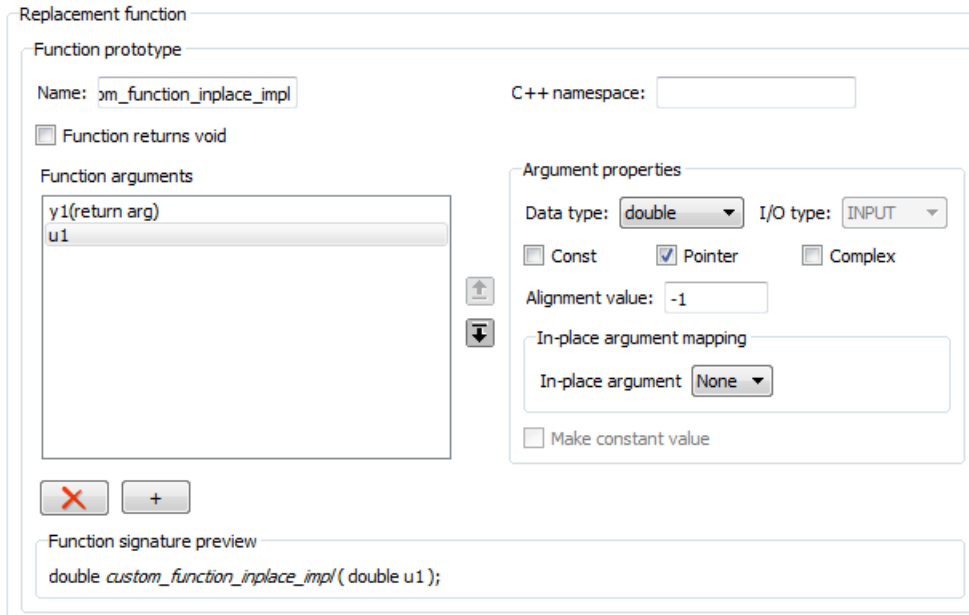
2. In the Code Replacement Tool, add a new table, select that table, and add a new function entry. For more information, see “Define Code Replacement Mappings” on page 52-30.
3. On the **Mapping Information** tab, select `Custom` for the **Function** parameter.
4. In the **function-name** text box, name the custom function. For this example, type the name `customFunction`.

5 Under the **Conceptual arguments** list box, click **+** to add two arguments. By default, the tool creates an output argument *y1* and an input argument *u1*, both of type `double`.

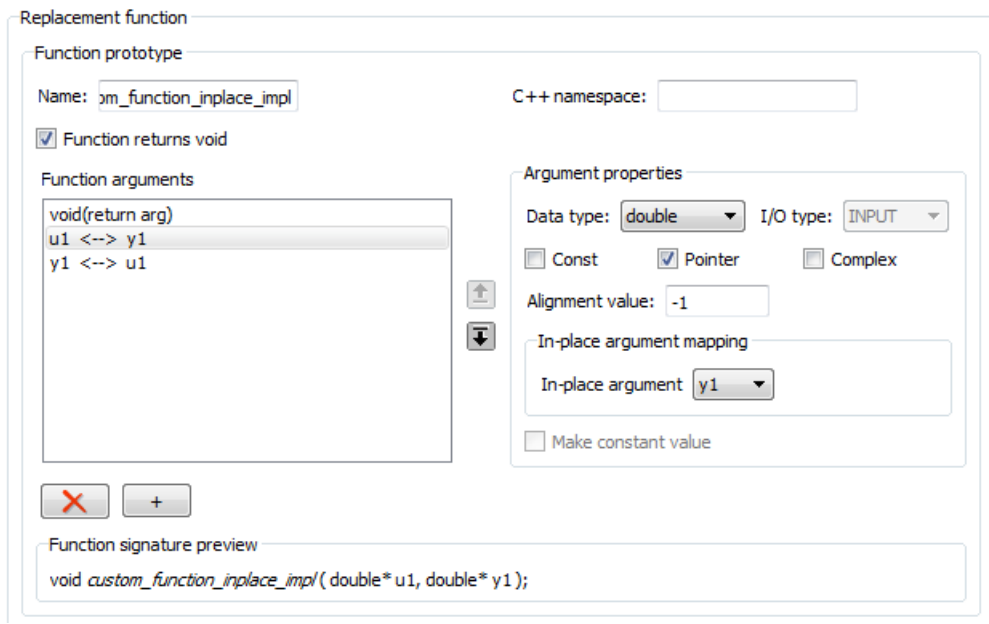
6 In the **Replacement function** > **Function prototype** section, type the name `custom_function_inplace_impl` in the **Name** text box.

7 Under the **Function arguments** list box, click **+** to add two function implementation arguments. By default, the tool creates an output argument *y1* and an input argument *u1*, both of type `double`.

8 For each input argument that you want to specify as in-place with a corresponding output argument, in the **Argument properties** box, select the **Pointer** check box. The **Argument properties** section of the dialog box expands to include an **In-place argument** drop-down list. For this example, in the **Function arguments** list, select input argument *u1*, and then select the **Pointer** check box.



9 From the **In-place argument** list, select *y1*, the output argument for the code replacement mapping. The **Function arguments** list box is updated to show possible in-place argument mappings.



10 Select and delete one of the two possible argument mappings. For this example, delete the mapping `y1<-->u1`.

11 In the **Function signature preview** box, if the function signature appears as expected, click **Apply**. Otherwise, make adjustments, and then click **Apply**. The function signature for this example, appears as

```
void custom_function_inplace_impl(double* u1);
```

12 Click **Validate entry**.

13 Save the code replacement table in the same folder as `customFunction.m`. Name the file `htfl_inplace_table.m`.

To test the example:

1 Register the table that contains the entry in a code replacement library.

2 Configure the code generator to use a code replacement library and to include the Code Replacements Report in the code generation report.

3 Generate the replacement code and a code generation report.

4 Review the code replacements.

Programmatic Argument Replacement Specification

This example shows how to specify in-place function argument replacement when replacing code for a MATLAB function programmatically. For the input implementation argument that shares the memory buffer, the example:

- Sets the name of the implementation argument to the same name as the corresponding conceptual argument.
- Associates the corresponding implementation argument with the argument property `ArgumentForInPlaceUse`.

1. Create the following MATLAB function, `customFunction.m`.

```
function y = customFunction(x)
% Function that updates the input and returns it as an output

coder.replace('-errorifnoreplacement');
x = sin(x);
```

2. Create a table definition file that contains a function definition. For example:

```
function hTable = crl_table_inplace()
```

3. Within the function body, create the table by calling the function `RTW.TflTable`.

```
hTable = RTW.TflTable;
```

4. Create an entry for the function mapping with a call to the `RTW.TflCFunctionEntry` function.

```
hEnt = RTW.TflCFunctionEntry;
```

5. Set function entry parameters with a call to the `setTflCFunctionEntryParameters` function.

```
setTflCFunctionEntryParameters(hEnt, ...
    'Key', 'customFunction', ...
    'Priority', 100, ...
    'ImplementationName', 'custom_function_inplace_impl', ...
    'SideEffects', true);
```

6. Create conceptual arguments `y1` and `u1`. This example uses calls to the `getTflArgFromString` and `addConceptualArg` functions to create and add the arguments.

```
arg = getTflArgFromString(hEnt, 'y1','double');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEnt, arg);
```

```
arg = getTflArgFromString(hEnt, 'u1','double');
addConceptualArg(hEnt, arg);
```

7 Create the implementation arguments and add them to the entry. This example uses calls to the `getTflArgFromString` function to create implementation arguments that map to arguments in the replacement function prototype: output argument *y1* and input argument *u1*. For each argument, the example uses the convenience method `setReturn` or `addArgument` to specify whether an argument is a return value or argument. For each argument, this example adds the argument to the entry array of implementation arguments.

```
arg = getTflArgFromString(hEnt, 'y2','void');
arg.IOType = 'RTW_IO_OUTPUT';
hEnt.Implementation.setReturn(arg);

arg = getTflArgFromString(hEnt, 'u1','double*');
arg.ArgumentForInPlaceUse = 'y1';
hEnt.Implementation.addArgument(arg);
```

8 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hLib, hEnt);
```

9 Save the table definition file. Use the name of the table definition function to name the file.

To test the example:

1 Register the table that contains the entry in a code replacement library.

2 Configure the code generator to use a code replacement library and to include the Code Replacements Report in the code generation report.

3 Generate the replacement code and a code generation report.

4 Review the code replacements.

More About

- “Code You Can Replace from MATLAB Code” on page 52-5
- “Define Code Replacement Mappings” on page 52-30
- “Develop a Code Replacement Library” on page 52-15

Data Alignment for Code Replacement

Code replacement libraries can align data objects passed into a replacement function to a specified boundary.

Code Replacement Data Alignment

You can take advantage of function implementations that require aligned data to optimize application performance when using MATLAB Coder. To configure data alignment for a function implementation:

1. Specify the data alignment requirements in a code replacement entry. Specify alignment separately for each implementation function argument or collectively for all function arguments. See "Specify Data Alignment Requirements for Function Arguments" on page 51-133.
2. Specify the data alignment capabilities and syntax for one or more compilers. Include the alignment specifications in a library registration entry in the `rtwTargetInfo.m` file. See "Provide Data Alignment Specifications for Compilers" on page 51-135.
3. Register the library containing the table entry and alignment specification object.
4. Configure the code generator to use the code replacement library and generate code. Observe the results.

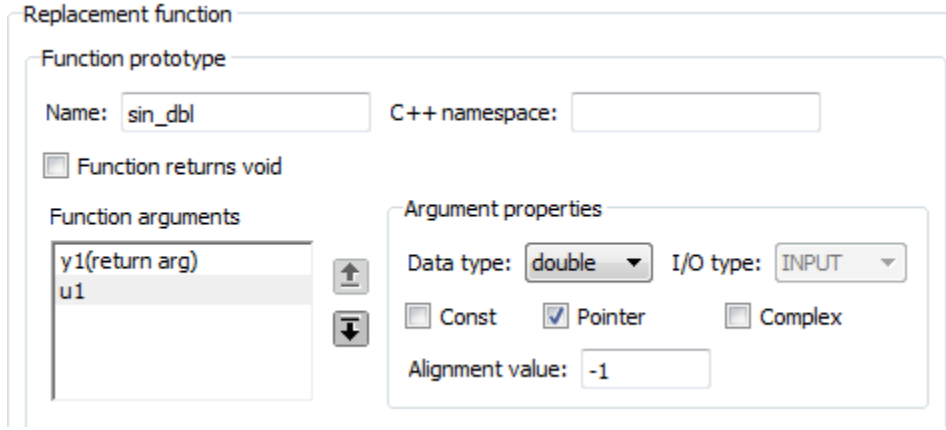
For examples, see "Basic Example of Code Replacement Data Alignment" on page 51-139 and the "Data Alignment for Function Implementations" section of the "Optimize Generated Code By Developing and Using Code Replacement Libraries - Simulink®" example page.

Note If replacement that requires alignment uses imported data (for example, I/O of an entry-point function or exported function), specify the data alignment with `coder.dataAlignment` statements in the MATLAB code. Specify alignment separately for each instance of imported data. See "Specify Data Alignment in MATLAB Code for Imported Data" on page 52-98.

Specify Data Alignment Requirements for Function Arguments

To specify the data alignment requirement for an argument in a code replacement entry:

- If you are defining a replacement function in a code replacement table registration file, create an argument descriptor object (`RTW.ArgumentDescriptor`). Use its `AlignmentBoundary` property to specify the required alignment boundary and assign the object to the argument `Descriptor` property.
- If you are defining a replacement function using the Code Replacement Tool, on the **Mapping Information** tab, in the **Argument properties** section for the replacement function, enter a value for the **Alignment value** parameter.



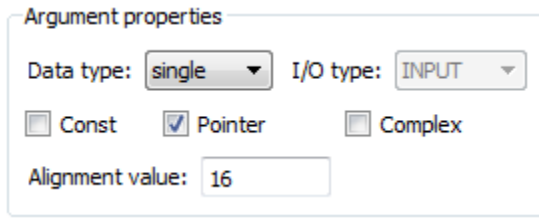
The `AlignmentBoundary` property (or **Alignment value** parameter) specifies the alignment boundary for data passed to a function argument, in number of bytes. The `AlignmentBoundary` property is valid only for addressable objects, including matrix and pointer arguments. It is not applicable for value arguments. Valid values are:

- `-1` (default) — If the data is a `Simulink.Bus`, `Simulink.Signal`, or `Simulink.Parameter` object, specifies that the code generator determines an optimal alignment based on usage. Otherwise, specifies that there is not an alignment requirement for this argument.
- Positive integer that is a power of 2, not exceeding 128 — Specifies number of bytes in the boundary. The starting memory address for the data allocated for the function argument is a multiple of the specified value. If you specify an alignment boundary that is less than the natural alignment of the argument data type, the alignment directive is emitted in the generated code. However, the target compiler ignores the directive.

The following code specifies the `AlignmentBoundary` for an argument as 16 bytes.

```
hLib = RTW.TflTable;
entry = RTW.TflCOperationEntry;
arg = getTflArgFromString(hLib, 'u1','single*');
desc = RTW.ArgumentDescriptor;
desc.AlignmentBoundary = 16;
arg.Descriptor = desc;
entry.Implementation.addArgument(arg);
```

The equivalent alignment boundary specification in the Code Replacement Tool dialog box is in this figure.



Note: If your model imports `Simulink.Bus`, `Simulink.Parameter`, or `Simulink.Signal` objects, specify an alignment boundary in the object properties, using the **Alignment** property. For more information, see `Simulink.Bus`, `Simulink.Parameter`, and `Simulink.Signal`.

Provide Data Alignment Specifications for Compilers

To support data alignment in generated code, describe the data alignment capabilities and syntax for your compilers in the code replacement library registration. Provide one or more alignment specifications for each compiler in a library registry entry.

To describe the data alignment capabilities and syntax for a compiler:

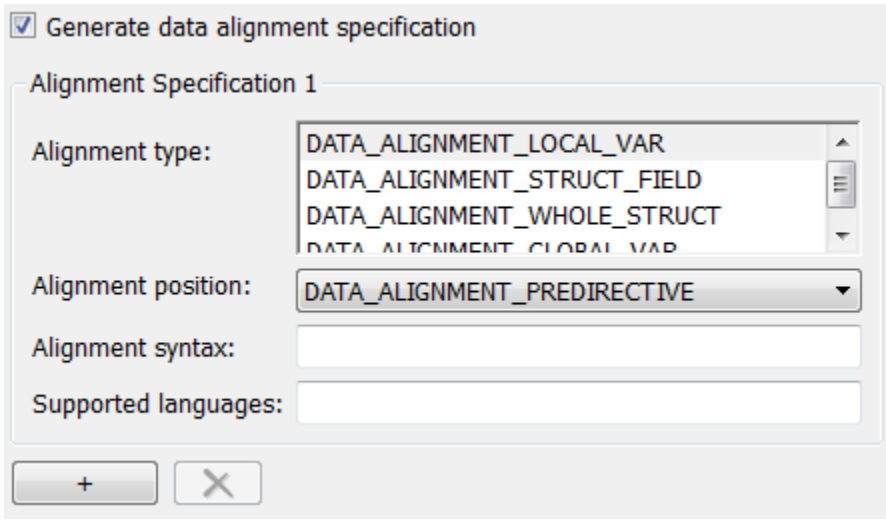
- If you are defining a code replacement library registration entry in a `rtwTargetInfo.m` customization file, add one or more `AlignmentSpecification` objects to an `RTW.DataAlignment` object. Attach the `RTW.DataAlignment` object to the `TargetCharacteristics` object of the registry entry.

  The `RTW.DataAlignment` object also has the property `DefaultMallocAlignment`, which specifies the default alignment boundary, in bytes, that the compiler uses for dynamically allocated memory. If the code generator uses dynamic memory allocation

for a data object involved in a code replacement, this value determines if the memory satisfies the alignment requirement of the replacement. If not, the code generator does not use the replacement. The default value for `DefaultMallocAlignment` is `-1`, indicating that the default alignment boundary used for dynamically allocated memory is unknown. In this case, the code generator uses the natural alignment of the data type to determine whether to allow a replacement.

Additionally, you can specify the alignment boundary for complex types by using the `addComplexTypeAlignment` function.

- If you are generating a customization file function using the Code Replacement Tool, fill out the following fields for each compiler.



Click the plus (+) symbol to add additional compiler specifications.

For each data alignment specification, provide the following information.

`AlignmentSpecification` Property	**Dialog Box Parameter**	**Description**
`AlignmentType`	**Alignment type**	Cell array of predefined enumerated strings, specifying which types of alignment this specification supports. • `DATA_ALIGNMENT_LOCAL_VAR` — Local variables.

Alignment-Specification Property	Dialog Box Parameter	Description
		• DATA_ALIGNMENT_GLOBAL_VAR — Global variables. • DATA_ALIGNMENT_STRUCT_FIELD — Individual structure fields. • DATA_ALIGNMENT_WHOLE_STRUCT — Whole structure, with padding (individual structure field alignment, if specified, is favored and takes precedence over whole structure alignment). Each alignment specification must specify at least DATA_ALIGNMENT_GLOBAL_VAR and DATA_ALIGNMENT_STRUCT_FIELD.

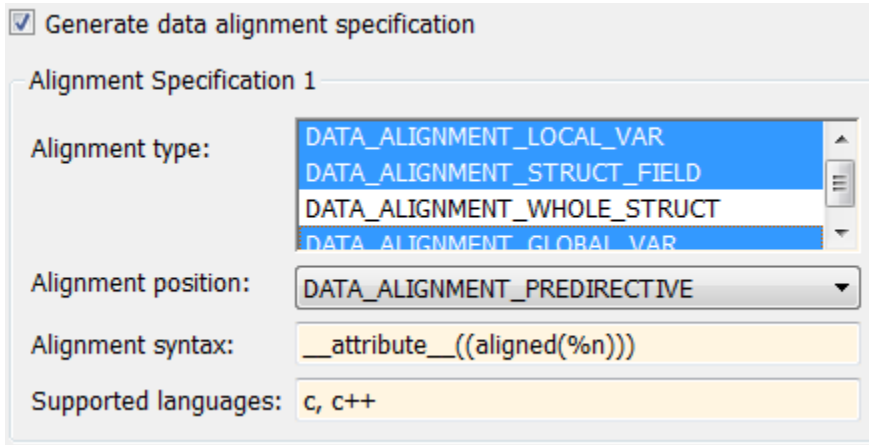
Alignment-Specification Property	Dialog Box Parameter	Description
AlignmentPosition	**Alignment position**	Predefined enumerated string specifying the position in which you must place the compiler alignment directive for alignment type DATA_ALIGNMENT_WHOLE_STRUCT: • DATA_ALIGNMENT_PREDIRECTIVE — The alignment directive is emitted before struct st_tag{…}, as part of the type definition statement (for example, MSVC). • DATA_ALIGNMENT_POSTDIRECTIVE — The alignment directive is emitted after struct st_tag{…}, as part of the type definition statement (for example, gcc). • DATA_ALIGNMENT_PRECEDING_STATEMENT — The alignment directive is emitted as a standalone statement immediately preceding the definition of the structure type. A semicolon (;) must terminate the registered alignment syntax. • DATA_ALIGNMENT_FOLLOWING_STATEMENT — The alignment directive is emitted as a standalone statement immediately following the definition of the structure type. A semicolon (;) must terminate the registered alignment syntax. For alignment types other than DATA_ALIGNMENT_WHOLE_STRUCT, code generation uses alignment position DATA_ALIGNMENT_PREDIRECTIVE.

Alignment-Specification Property	Dialog Box Parameter	Description
AlignmentSyntax-Template	**Alignment syntax**	Specifies the alignment directive string that the compiler supports. The string is registered as a syntax template that has placeholders in it. These placeholders are supported: • `%n` — Replaced by the alignment boundary for the replacement function argument. • `%s` — Replaced by the aligned symbol, usually the identifier of a variable. For example, for the gcc compiler, you can specify `__attribute__((aligned(%n)))`, or for the MSVC compiler, `__declspec(align(%n))`.
SupportedLanguage	**Supported languages**	Cell array specifying the languages to which this alignment specification applies, among `c` and `c++`. Sometimes alignment syntax and position differ between languages for a compiler. .

Here is a data alignment specification for the GCC compiler:

```
da = RTW.DataAlignment;

as = RTW.AlignmentSpecification;
as.AlignmentType = {'DATA_ALIGNMENT_LOCAL_VAR', ...
                    'DATA_ALIGNMENT_STRUCT_FIELD', ...
                    'DATA_ALIGNMENT_GLOBAL_VAR'};
as.AlignmentSyntaxTemplate = '__attribute__((aligned(%n)))';
as.AlignmentPosition = 'DATA_ALIGNMENT_PREDIRECTIVE';
as.SupportedLanguages = {'c', 'c++'};
da.addAlignmentSpecification(as);

tc = RTW.TargetCharacteristics;
tc.DataAlignment = da;
```

Here is the corresponding specification in the **Generate customization** dialog box of the Code Replacement Tool.



Specify Data Alignment in MATLAB Code for Imported Data

If MATLAB Coder code replacements that require data alignment use imported data, such as an entry-point or exported function I/O, specify data alignment to external code with `coder.dataAlignment` statements in the MATLAB code.

If MATLAB Coder code replacements occur that require data alignment (uses imported data), such as an entry-point or exported function with I/O, specify code replacement data alignment with `coder.DataAlignment` statements in the MATLAB code.

To specify the data alignment requirements for imported data in a MATLAB code:

- For each instance of imported data that requires data alignment, specify the alignment in the function with a `coder.dataAlignment` statement of the form:

  ```
  coder.dataAlignment('varName', align_value)
  ```

- The *`varName`* is a character array of the variable name that requires alignment information specification. The *`align_value`* is an integer number which should be a power of 2, from 2 through 128. This number specifies the power-of-2 byte alignment boundary.
- An example function that specifies data alignment is:

  ```
  function y = testFunction(x1,x2)

  coder.dataAlignment('x1',16);   % Specifies information
  ```

```
coder.dataAlignment('x2',16);    % Specifies information
coder.dataAlignment('y',16);     % Specifies information

y = x1 + x2;

end
```

If `testFunction` is an entry-point or exported function, imported data `x1`, `x2`, and `y` are not aligned automatically by the code generator. The `coder.DataAlignment` statements for these variables are only meant as information for the code generator. The call sites allocating memory for the data need to ensure that the data is aligned as specified.

You also can specify code replacement data alignment for exported data, such as a `global` variable or an `ExportedGlobal` custom storage class. For more information, see "Introduction to Custom Storage Classes" on page 23-2.

Replacing Math Functions and Operators with Implementations that require Data Alignment - MATLAB®

This example shows how to develop and use code replacement library entries for target-specific function implementations that require data to be aligned to optimize application performance. To configure data alignment for a function implementation:

- Specify the data alignment requirements in a table entry. You can specify alignment for implementation function arguments individually or collectively.
- Specify the data alignment capabilities and syntax for your compiler. Attach an AlignmentSpecification object to the TargetCharacteristics object of the registry entry specified in your rtwTargetInfo.m file.

If externally allocated data (e.g. entry-point function arguments) are used in an operation that can be replaced with an implementation that requires alignment, use the coder.dataAlignment directive to specify alignment so that replacement occurs.

This example is configured to use either the GCC, Clang, or MSVC compilewhosrs.

Create a New Folder and Copy Relevant Files

The following code creates a folder in your current working folder (pwd). The new folder will contain the files that are relevant for this example. If you do not want to affect the

current folder (or if you cannot generate files in this folder), you should change your working folder.

Run Command: Create a New Folder and Copy Relevant Files

```
coderdemo_setup('coderdemo_crlalign');
cleanupObj = {};
mlpath = addpath(fullfile(matlabroot,'toolbox','coder','codegendemos','coderdemo_crlali
cleanupObj{end+1} = onCleanup(@()path(mlpath));
```

Check selected compiler

This example is configured to use either GCC, Clang, or MSVC to compile the generated code.

```
cc = rtwprivate('getMexCompilerInfo');
isDaDemoSupported = strcmpi(cc.comp.Manufacturer,'GNU') || ...
                    strcmpi(cc.comp.Manufacturer,'Apple') || ...
                    strcmpi(cc.comp.Manufacturer,'Microsoft');
if ~isDaDemoSupported
  recMsg = ['Use "mex -setup" to select either GCC, Clang, or MSVC and restart this '
            'example'];
  warning(['Example %s is configured to use either GCC, Clang, or MSVC to compile ' ..
           'the generated code. %s.'], mfilename,recMsg);
end
```

Set MATLAB Coder options

Set up the configuration object and define the function input types.

```
cfg = coder.config('lib','ecoder',true);
cfg.GenerateReport = false;
cfg.LaunchReport = false;
cfg.VerificationMode = 'SIL';
cfg.CodeExecutionProfiling = true;

len = 400000;
args = {coder.typeof(single(0),[len,1]), ...
        coder.typeof(single(0))};
global g1;
g1 = single(zeros([len,1]));
```

Generate code using the SIMD Examples Code Replacement Library

To see the code replacement table definition file, look here.

```
RTW.TargetRegistry.getInstance('reset');

mcode_da16 = 'biased_sum_of_square_differences_da16';
cfg.CodeReplacementLibrary = 'SIMD Examples';
codegen('-config',cfg, mcode_da16,'-args',args,'-global',{'g1',g1});
```

Inspect the MATLAB Coder Generated Code

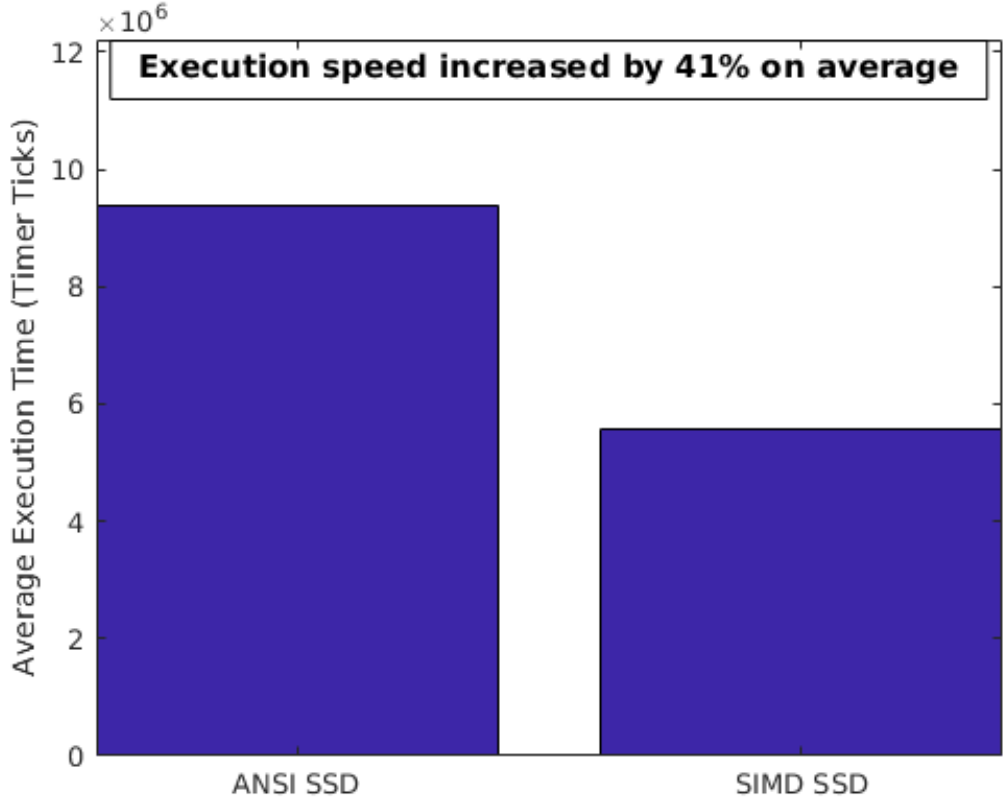
After compiling, you may want to explore the generated source code.
matlab:edit(fullfile(pwd,'codegen','lib','biased_sum_of_square_differences_da16','sum_of_square_differ

Performance Gain from Data Alignment

Compare performance of normal ANSI code against the earlier generated code that used SIMD intrinsics.

```
% Generate ansi code
mcode_noda = 'biased_sum_of_square_differences';
cfg.CodeReplacementLibrary = 'None';
codegen('-config',cfg, mcode_noda,'-args',args,'-global',{'g1',g1});

% Run SIMD executable and collect execution profile information
coder.runTest('run_biased_ssd_da16',[mcode_da16,'_sil.',mexext])
pause(120);
clear([mcode_da16,'_sil']); % stop simulation
executionProfile_simd = getCoderExecutionProfile(mcode_da16);
idx_section = find(strcmp(mcode_da16,{executionProfile_simd.Sections.Name}),1);
avg_selftime_simd = executionProfile_simd.Sections(idx_section).TotalSelfTimeInTicks/ex

% Run ANSI executable and collect execution profile information
coder.runTest('run_biased_ssd',[mcode_noda,'_sil.',mexext])
pause(120);
clear([mcode_noda,'_sil']); % stop simulation
executionProfile_ansi = getCoderExecutionProfile(mcode_noda);
idx_section = find(strcmp(mcode_noda,{executionProfile_ansi.Sections.Name}),1);
avg_selftime_ansi = executionProfile_ansi.Sections(idx_section).TotalSelfTimeInTicks/ex

% Compare execution profile results
barObj = bar([avg_selftime_ansi; ...
    avg_selftime_simd]);
axesObj = barObj.Parent;
figObj = axesObj.Parent;
axesObj.XTickLabel = {'ANSI SSD', 'SIMD SSD'};
axesObj.YLabel.String = 'Average Execution Time (Timer Ticks)';
```

```
axesObj.YLim = [min([0,avg_selftime_ansi,avg_selftime_simd]), ...
                max(avg_selftime_ansi,avg_selftime_simd)*1.3];

percent_perf_gain = 100 * (avg_selftime_ansi-avg_selftime_simd)/avg_selftime_ansi;
annotation(figObj, 'textbox',axesObj.Position, ...
    'String',sprintf('Execution speed increased by %d%% on average',percent_perf_gain),
    'FontWeight', 'bold', 'FontSize', 12, 'HorizontalAlignment', 'center', ...
    'FitBoxToText','On');

### Starting SIL execution for 'biased_sum_of_square_differences_da16'
    To terminate execution: <a href="matlab: targets_hyperlink_manager('run',1);">clear
    Execution profiling data is available for viewing. Open <a href="matlab:Simulink.sc
    Execution profiling report available after termination.
### Stopping SIL execution for 'biased_sum_of_square_differences_da16'
    Execution profiling report: <a href="matlab: targets_hyperlink_manager('run',3);">r

### Starting SIL execution for 'biased_sum_of_square_differences'
    To terminate execution: <a href="matlab: targets_hyperlink_manager('run',4);">clear
    Execution profiling data is available for viewing. Open <a href="matlab:Simulink.sc
    Execution profiling report available after termination.
### Stopping SIL execution for 'biased_sum_of_square_differences'
    Execution profiling report: <a href="matlab: targets_hyperlink_manager('run',6);">r
```



Cleanup

Remove files and return to original folder

Run Command: Cleanup

```
RTW.TargetRegistry.getInstance('reset');
cleanup
```

More About

- “Code You Can Replace From Simulink Models” on page 51-7

- “Define Code Replacement Mappings” on page 51-42
- “Develop a Code Replacement Library” on page 51-27

Replace MATLAB Functions with Custom Code Using `coder.replace`

The `coder.replace` function provides the ability to replace a specified MATLAB function with a code replacement function in generated code. Use `coder.replace` in MATLAB code from which you want to generate C code using:

- MATLAB Coder
- MATLAB code in a Simulink MATLAB Function block

You can replace MATLAB functions that have:

- Single or multiple inputs
- Single or multiple outputs
- Scalar and matrix inputs and outputs

Supported types include:

- `single`, `double` (complex and noncomplex)
- `int8`, `uint8` (complex and noncomplex)
- `int16`, `uint16` (complex and noncomplex)
- `int32`, `uint32` (complex and noncomplex)
- Fixed-point integers
- Mixed types (different type on each input)

More About

- "Code You Can Replace from MATLAB Code" on page 52-5
- "Define Code Replacement Mappings" on page 52-30
- "Develop a Code Replacement Library" on page 52-15

Replace `coder.ceval` Calls to External Functions

The `coder.ceval` function calls external C/C++ functions from code generated from MATLAB code. The code replacement software supports replacement of the function that you specify in a call to `coder.ceval`. An application of this code replacement scenario is to write generic MATLAB code that you can customize for different platforms with code replacements. A code replacement library can define hardware-specific code replacements for the function call. Use `coder.ceval` in MATLAB code from which you want to generate C code using:

- MATLAB Coder
- MATLAB code in a Simulink MATLAB Function block

Example Files

For the examples in "Interactive External Function Call Replacement Specification with Code Replacement Tool" on page 52-107 and "Programmatic External Function Call Replacement Specification" on page 52-108 you must have set up the following:

- Custom C function `my_add.c`.

```
/* my_add.c */

#include "my_add.h"

double my_add(double in1, double in2)
{
  return in1 + in2;
}
```

- Custom C header file `my_add.h`.

```
/* my_add.h */

double my_add(double in1, double in2);
```

- MATLAB function `call_my_add.m`, which uses `coder.ceval` to invoke `my_add.c`.

```
function y = call_my_add(in1, in2)  %#codegen

y=0.0;

if ~coder.target('Rtw')
% Executing in MATLAB, call MATLAB equivalent of C function my_add
```

```
  y= in1+in2;
else
% Executing in generated code, call C function my_add
  y = coder.ceval('my_add', in1, in2);
end
```

- MATLAB test function `call_my_add_test.m`, which calls `call_my_add.m`.

```
in1=10;
in2=20;

y = call_my_add(in1, in2);

disp('Output')
disp('y =')
disp(y);
```

- Replacement C function `my_add_replacement.c`.

```
/* my_add_replacement.c */

#include "my_add_replacement.h"

double my_add_replacement(double in1, double in2)
{
  return in1 + in2;
}
```

- Replacement C header file `my_add_replacement.h`.

```
/* my_add_replacement.h */

double my_add_replacement(double in1, double in2);
```

Interactive External Function Call Replacement Specification with Code Replacement Tool

This example shows how to define a code replacement table entry for a MATLAB function that calls `coder.ceval` to invoke an external C function. The example shows how to define the entry interactively with the Code Replacement Tool.

1. Identify or create the C/C++ code and relevant header files, the MATLAB function that calls `coder.ceval`, a MATLAB test function, and the source and header files for your replacement code. To follow along with this example, set up the files identified in “Example Files” on page 52-106.

2. In the Code Replacement Tool, add a table, select that table, and add a function entry. For more information, see "Define Code Replacement Mappings" on page 52-30.
3. On the **Mapping Information** tab, select `Custom` for the **Function** parameter.
4. In the **function-name** text box, type the custom function name. For this example, type the name `my_add`.
5. Under the **Conceptual arguments** list box, click **+** to add three arguments. By default, the tool creates an output argument `y1` and input arguments `u1` and `u2` of type `double`.
6. In the **Replacement function** > **Function prototype** section, type the name `my_add_replacement` in the **Name** text box.
7. Under the **Function arguments** list box, click **+** to add three function implementation arguments. By default, the tool creates an output argument `y1` and input arguments `u1` and `u2` of type double. Use the default settings.
8. In the **Function signature preview** box, if you see the expected function signature, click **Apply**. The function signature for this example, appears as:

   ```
   double my_add_replacement(double u1, double u2);
   ```
9. On the **Build Information** tab, specify `my_add_replacement.h` for the **Implementation header file** parameter and `my_add_replacement.c` for the **Implementation source file**.
10. Click **Validate entry**.
11. Save the code replacement table in the same folder as `my_add_replacement.c`. Name the file `crl_table_my_add.m`.

To test the example:

1. Register the table that contains the entry in a code replacement library.
2. Configure the code generator to use the code replacement library and to include the Code Replacements Report in the code generation report.
3. Generate code and the report.
4. Review the code replacements.

Programmatic External Function Call Replacement Specification

This example shows how to define a code replacement table entry for a MATLAB function that calls `coder.ceval` to invoke an external C function. The example shows how to define the entry programmatically.

1. Identify or create the C/C++ code and relevant header files, the MATLAB function that calls `coder.ceval` to invoke the C/C++ function, a MATLAB test function, and the source and header files for your replacement code. To follow along with this example, set up the files identified in "Example Files" on page 52-106.
2. Create a table definition file that contains a function definition. For example:

   ```
   function hLib = crl_table_my_add
   ```
3. Within the function body, create the table by calling the function `RTW.TflTable`.
4. Create an entry for the function mapping with a call to the `RTW.TflCFunctionEntry` function.

   ```
   hEnt = RTW.TflCFunctionEntry;
   ```
5. Set function entry parameters with a call to the `setTflCFunctionEntryParameters` function.

   ```
   hEnt.setTflCFunctionEntryParameters( ...
             'Key', 'my_add', ...
             'Priority', 100, ...
             'ImplementationName', 'my_add_replacement', ...
             'ImplementationHeaderFile', 'my_add_replacement.h', ...
             'ImplementationSourceFile', 'my_add_replacement.c');
   ```
6. Create conceptual arguments `y1`, `u1`, and `u1`. This example uses calls to the `getTflArgFromString` and `addConceptualArg` functions to create and add the arguments.

   ```
   arg = hEnt.getTflArgFromString('y1','double');
   arg.IOType = 'RTW_IO_OUTPUT';
   hEnt.addConceptualArg(arg);

   arg = hEnt.getTflArgFromString('u1','double');
   hEnt.addConceptualArg(arg);

   arg = hEnt.getTflArgFromString('u2','double');
   hEnt.addConceptualArg(arg);
   ```
7. Create the implementation arguments and add them to the entry. This example uses calls to the `getTflArgFromString` function to create implementation arguments. These functions map to arguments in the replacement function prototype: output argument `y1` and input arguments `u1` and `u2`. For each argument, the example uses the convenience method `setReturn` or `addArgument` to specify whether an argument is a return value or argument. For each argument, this example adds the argument to the entry array of implementation arguments.

```
arg = hEnt.getTflArgFromString('y1','double');
arg.IOType = 'RTW_IO_OUTPUT';
hEnt.Implementation.setReturn(arg);

arg = hEnt.getTflArgFromString('u1','double');
hEnt.Implementation.addArgument(arg);

arg = hEnt.getTflArgFromString('u2','double');
hEnt.Implementation.addArgument(arg);
```

8 Add the entry to a code replacement table with a call to the `addEntry` function.

```
hLib.addEntry(hEnt);
```

9 Save the table definition file. Use the name of the table definition function to name the file.

To test the example:

1 Register the table that contains the entry in a code replacement library.

2 Configure the code generator to use the code replacement library and to include the Code Replacements Report in the code generation report.

3 Generate code and the report.

4 Review the code replacements.

More About

- “Code You Can Replace from MATLAB Code” on page 52-5
- “Define Code Replacement Mappings” on page 52-30
- “Develop a Code Replacement Library” on page 52-15

Reserved Identifiers and Code Replacement

The code generator and C programming language use, internally, reserved keywords for code generation. Do not use reserved keywords as identifiers or function names. Reserved keywords for code generation include many code replacement library identifiers, the majority of which are function names, such as `acos`.

To view a list of reserved identifiers for the code replacement library that you use to generate code, specify the name of the library in a call to the function `RTW.TargetRegistry.getInstance.getTflReservedIdentifiers`. For example:

```
crl_ids = RTW.TargetRegistry.getInstance.getTflReservedIdentifiers('GNU99 (GNU)')
```

In a code replacement table, the code generator registers each function implementation name defined by a table entry 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 code replacement reserved identifiers, use the `setReservedIdentifiers` function. This function registers specified reserved identifiers to be associated with a code replacement table.

You can register up to four reserved identifier structures in a code replacement table. You can associate one set of reserved identifiers with a code replacement library, while the other three (if present) must be associated with ANSI C. The following example shows a reserved identifier structure that specifies two identifiers and the associated header file.

```
d{1}.LibraryName = 'ANSI_C';
d{1}.HeaderInfos{1}.HeaderName = 'math.h';
d{1}.HeaderInfos{1}.ReservedIds = {'y0', 'y1'};
```

The code generator adds the identifiers to the list of reserved identifiers and honors them during the build procedure.

More About

- “Code You Can Replace from MATLAB Code” on page 52-5
- “Customize Match and Replacement Process” on page 52-112
- “Define Code Replacement Mappings” on page 52-30
- “Develop a Code Replacement Library” on page 52-15

Customize Match and Replacement Process

During the build process, the code generator uses:

- Preset match criteria to identify functions and operators for which application-specific implementations replace default implementations.
- Preset replacement function signatures.

It is possible that preset match criteria and preset replacement function signatures do not completely meet your function and operator replacement needs. For example:

- You want to replace an operator with a particular fixed-point implementation function only when fraction lengths are within a particular range.
- When a match occurs, you want to modify your replacement function signature based on compile-time information, such as passing fraction-length values into the function.

To add extra logic into the code replacement match and replacement process, create custom code replacement table entries. With custom entries, you can specify additional match criteria and modify the replacement function signature to meet application needs.

To create a custom code replacement entry:

1. Create a custom code replacement entry class, derived from `RTW.TflCFunctionEntryML (for function replacement) or RTW.TflCOperationEntryML` (for operator replacement).
2. In your derived class, implement a `do_match` method with a fixed preset signature as a MATLAB function. In your `do_match` method, provide either or both of the following customizations that instantiate the class:

    - Add match criteria that the base class does not provide. The base class provides a match based on:

        - Argument number
        - Argument name
        - Signedness
        - Word size
        - Slope (if not specified with wildcards)
        - Bias (if not specified with wildcards)
        - Math modes, such as saturation and rounding

    - Operator or function key
  - Modify the implementation signature by adding additional arguments or setting constant input argument values. You can inject a constant value, such as an input scaling value, as an additional argument to the replacement function.
3. Create code replacement entries that instantiate the custom entry class.
4. Register a library containing the code replacement table that includes your entries.

During code generation, the code replacement match process tries to match function or operator call sites with the base class of your derived entry class. If the process finds a match, the software calls your `do_match` method to execute your additional match logic (if any) and your replacement function customizations (if any).

Customize Match and Replacement Process for Operators

This example shows how to create custom code replacement entries that add logic to the code match and replacement process for a scalar operation. Custom entries specify additional match criteria or modify the replacement function signature to meet application needs.

For example:

- When fraction lengths are within a specific range, replace an operator with a fixed-point implementation function.
- When a match occurs, modify the replacement function signature based on compile-time information, such as passing fraction-length values into the function.

This example modifies a fixed-point addition replacement such that the implementation function passes in the fraction lengths of the input and output data types as arguments.

To create custom code replacement entries that add logic to the code replacement match and replacement process:

1. Create a class, for example `TflCustomOperationEntry`, that is derived from the base class `RTW.TflCOperationEntryML`. The derived class defines a `do_match` method with the following signature:

```
function ent = do_match(hThis, ...
        hCSO, ...
        targetBitPerChar, ...
        targetBitPerShort, ...
        targetBitPerInt, ...
```

```
        targetBitPerLong, ...
        targetBitPerLongLong)
```

In the `do_match` signature:

- `ent` is the return handle, which is returned as empty (indicating that the match failed) or as a `TflCOperationEntry` handle.
- `hThis` is the handle to the class instance.
- `hCSO` is a handle to an object that the code generator creates for querying the library for a replacement.
- Remaining arguments are the number of bits for various data types of the current target.

The `do_match` method adds match criteria that the base class does not provide. The method makes modifications to the implementation signature. In this case, the `do_match` method relies on the base class for checking word size and signedness. `do_match` must match only the number of conceptual arguments to the value 3 (two inputs and one output) and the bias for each argument to value 0. If the code generator finds a match, `do_match`:

- Sets the return handle.
- Removes slope and bias wild cards from the conceptual arguments (the match is for specific slope and bias values).
- Writes fraction-length values for the inputs and output into replacement function arguments 3, 4, and 5.

You can create and add three additional implementation function arguments for passing fraction lengths in the class definition or in each code replacement entry definition that instantiates this class. This example creates the arguments, adds them to a code replacement table definition file, and sets them to specific values in the class definition code.

```
classdef TflCustomOperationEntry < RTW.TflCOperationEntryML
  methods
    function ent = do_match(hThis, ...
        hCSO, ... %#ok
        targetBitPerChar, ... %#ok
        targetBitPerShort, ... %#ok
        targetBitPerInt, ... %#ok
        targetBitPerLong, ... %#ok
        targetBitPerLongLong) %#ok

      % DO_MATCH - Create a custom match function. The base class
```

```
        % checks the types of the arguments prior to calling this
        % method. This class will check additional data and can
        % modify the implementation function.

        % The base class checks word size and signedness. Slopes and biases
        % have been wildcarded, so the only additional checking to do is
        % to check that the biases are zero and that there are only three
        % conceptual arguments (one output, two inputs)

        ent = []; % default the return to empty, indicating the match failed

        if length(hCSO.ConceptualArgs) == 3 && ...
            hCSO.ConceptualArgs(1).Type.Bias == 0 && ...
            hCSO.ConceptualArgs(2).Type.Bias == 0 && ...
            hCSO.ConceptualArgs(3).Type.Bias == 0

          % Modify the default implementation. Since this is a
          % generator entry, a concrete entry is created using this entry
          % as a template. The type of entry being created is a standard
          % TflCOperationEntry. Using the standard operation entry
          % provides required information, and you do not need
          % a custom match function.
          ent = RTW.TflCOperationEntry(hThis);

          % Since this entry is modifying the implementation for specific
          % fraction-length values (arguments 3, 4, and 5), the conceptual argument
          % wild cards must be removed (the wildcards were inherited from the
          % generator when it was used as a template for the concrete entry).
          % This concrete entry is now for a specific slope and bias.
          % hCSO holds the slope and bias values (created by the code generator).
          for idx=1:3
            ent.ConceptualArgs(idx).CheckSlope = true;
            ent.ConceptualArgs(idx).CheckBias = true;

            % Set the specific Slope and Biases
            ent.ConceptualArgs(idx).Type.Slope = hCSO.ConceptualArgs(idx).Type.Slope;
            ent.ConceptualArgs(idx).Type.Bias = 0;
          end

          % Set the fraction-length values in the implementation function.
          ent.Implementation.Arguments(3).Value = ...
             -1.0*hCSO.ConceptualArgs(2).Type.FixedExponent;
          ent.Implementation.Arguments(4).Value = ...
             -1.0*hCSO.ConceptualArgs(3).Type.FixedExponent;
          ent.Implementation.Arguments(5).Value = ...
             -1.0*hCSO.ConceptualArgs(1).Type.FixedExponent;
        end
      end
    end
  end
```

Exit the class folder and return to the previous working folder.

2 Create and save the following code replacement table definition file, `crl_table_custom_sinfcn_double.m`. This file defines a code replacement table

that contains a single operator entry, an entry generator for unsigned 32-bit fixed-point addition operations, with arbitrary fraction-length values on the inputs and the output. The table entry:

- Instantiates the derived class `TflCustomOperationEntry` from the previous step. If you want to replace word sizes and signedness attributes, you can use the same derived class, but not the same entry, because you cannot use a wild card with the `WordLength` and `IsSigned` arguments. For example, to support `uint8`, `int8`, `uint16`, `int16`, and `int32`, add five other distinct entries. To use different implementation functions for saturation and rounding modes other than overflow and round to floor, add entries for those match permutations.
- Sets operator entry parameters with the call to the `setTflCOperationEntryParameters` function.
- Calls the `createAndAddConceptualArg` function to create conceptual arguments `y1`, `u1`, and `u2`.
- Calls `createAndSetCImplementationReturn` and `createAndAddImplementationArg` to define the signature for the replacement function. Three of the calls to `createAndAddImplementationArg` create implementation arguments to hold the fraction-length values for the inputs and output. Alternatively, the entry can omit those argument definitions. Instead, the `do_match` method of the derived class `TflCustomOperationEntry` can create and add the three implementation arguments. When the number of additional implementation arguments required can vary based on compile-time information, use the alternative approach.
- Calls `addEntry` to add the entry to a code replacement table.

```
function hTable = crl_table_custom_add_ufix32

hTable = RTW.TflTable;

%% Add TflCustomOperationEntry
op_entry = TflCustomOperationEntry;

setTflCOperationEntryParameters(op_entry, ...
    'Key',                      'RTW_OP_ADD', ...
    'Priority',                 30, ...
    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes',            {'RTW_ROUND_FLOOR'}, ...
    'ImplementationName',       'myFixptAdd', ...
    'ImplementationHeaderFile', 'myFixptAdd.h', ...
```

```
    'ImplementationSourceFile', 'myFixptAdd.c');

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
     'Name',       'y1', ...
     'IOType',     'RTW_IO_OUTPUT', ...
     'CheckSlope', false, ...
     'CheckBias',  false, ...
     'DataType',   'Fixed', ...
     'Scaling',    'BinaryPoint', ...
     'IsSigned',   false, ...
     'WordLength', 32);

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
      'Name',       'u1', ...
      'IOType',     'RTW_IO_INPUT', ...
      'CheckSlope', false, ...
      'CheckBias',  false, ...
      'DataType',   'Fixed', ...
      'Scaling',    'BinaryPoint', ...
      'IsSigned',   false, ...
      'WordLength', 32);

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
       'Name',       'u2', ...
       'IOType',     'RTW_IO_INPUT', ...
       'CheckSlope', false, ...
       'CheckBias',  false, ...
       'DataType',   'Fixed', ...
       'Scaling',    'BinaryPoint', ...
       'IsSigned',   false, ...
       'WordLength', 32);

% Specify replacement function signature
createAndSetCImplementationReturn(op_entry, 'RTW.TflArgNumeric', ...
    'Name',       'y1', ...
    'IOType',     'RTW_IO_OUTPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric', ...
    'Name',       'u1', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
```

```
    'WordLength', 32, ...
    'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric', ...
    'Name',       'u2', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0);

% Add 3 fraction-length args. Actual values are set during code generation.
createAndAddImplementationArg(op_entry, 'RTW.TflArgNumericConstant', ...
    'Name',       'fl_in1', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value',       0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumericConstant', ...
    'Name',       'fl_in2', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value',       0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumericConstant', ...
    'Name',       'fl_out', ...
    'IOType',     'RTW_IO_INPUT', ...
    'IsSigned',   false, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value',       0);

addEntry(hTable, op_entry);
```

3 Check the validity of the operator entry.

- At the command prompt, invoke the table definition file.

  ```
  tbl = crl_table_custom_sinfcn_double
  ```

- In the Code Replacement Viewer, view the table definition file.

  ```
  crviewer(crl_table_custom_sinfcn_double)
  ```

More About

- "Code You Can Replace from MATLAB Code" on page 52-5
- "Define Code Replacement Mappings" on page 52-30
- "Develop a Code Replacement Library" on page 52-15

Scalar Operator Code Replacement

This example shows how to define a code replacement mapping for a scalar operator. The example defines a mapping for the + (addition) operator programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

1. Create a table definition file that contains a function definition. For example:

   ```
   function hTable = crl_table_add_uint8
   ```

2. Within the function body, create the table by calling the function `RTW.TflTable`.

   ```
   hTable = RTW.TflTable;
   ```

3. Create an entry for the operator mapping with a call to the `RTW.TflCOperationEntry` function.

   ```
   % Create operation entry
   op_entry = RTW.TflCOperationEntry;
   ```

4. Set function entry parameters with a call to the `setTflCOperationEntryParameters` function.

   ```
   % Define addition operation of built-in uint8 data type
   % Saturation on, Rounding unspecified
   setTflCOperationEntryParameters(op_entry, ...
                       'Key',                      'RTW_OP_ADD', ...
                       'Priority',                 90, ...
                       'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
                       'RoundingModes',            {'RTW_ROUND_UNSPECIFIED'}, ...
                       'ImplementationName',       'u8_add_u8_u8', ...
                       'ImplementationHeaderFile', 'u8_add_u8_u8.h', ...
                       'ImplementationSourceFile', 'u8_add_u8_u8.c' );
   ```

5. Create conceptual arguments `y1`, `u1`, and `u2`. There are multiple ways to set up the conceptual arguments. This example uses calls to the `getTflArgFromString` and `addConceptualArg` functions to create and add the arguments.

   ```
   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 );
   ```

6. Copy the conceptual arguments to the implementation arguments. There are multiple ways to set up the implementation arguments. This example uses a call

to the `copyConceptualArgsToImplementation` function to create and add implementation arguments to the entry by copying matching conceptual arguments.

```
copyConceptualArgsToImplementation(op_entry);
```

7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

8 Save the table definition file. Use the name of the table definition function to name the file.

More About

- "Code You Can Replace from MATLAB Code" on page 52-5
- "Define Code Replacement Mappings" on page 52-30
- "Remap Operator Output to Function Input" on page 52-143
- "Customize Match and Replacement Process" on page 52-112
- "Develop a Code Replacement Library" on page 52-15
- "What Is Code Replacement Customization?" on page 52-3

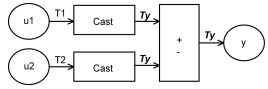
Addition and Subtraction Operator Code Replacement

Consider the following when defining mappings for addition and subtraction operator code replacements.

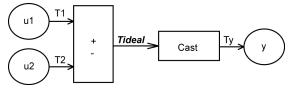
Algorithm Options

When creating a code replacement table entry for an addition or subtraction operator, first determine the type of algorithm that your library function implements.

- Cast-before-operation (CBO), default — Prior to performing the addition or subtraction operation, the algorithm type casts input values to the output type. If the output data type cannot exactly represent the input values, losses can occur as a result of the cast to the output type. Additional loss can occur when the result of the operation is cast to the final output type.



- Cast-after-operation (CAO) — The algorithm computes the ideal result of the addition or subtraction operation of the two inputs. The algorithm then type casts the result to the output data type. Loss occurs during the type cast. This algorithm behaves similarly to the C language except when the signedness of the operands does not match. For example, when you add a signed long operand to an unsigned long operand, standard C language rules convert the signed long operand to an unsigned long operand. The result is a value that is not ideal.



Interactive Specification with Code Replacement Tool

When you use the Code Replacement Tool to create a code replacement table entry for an addition or subtraction operation, the tool displays an **Algorithm** menu. Use that menu to specify the `Cast before operation` or `Cast after operation` algorithm for that entry.

Programmatic Specification

Create a code replacement table file, as a MATLAB function, that describes the addition or subtraction code replacement table entry. In the call to `setTflCOperationEntryParameters`, set at least these parameters:

- `Key` to `RTW_OP_ADD` or `RTW_OP_MINUS`
- `ImplementationName` to the name of your replacement function
- `EntryInfoAlgorithm` to `RTW_CAST_BEFORE_OP` (cast-before-operation) or `RTW_CAST_AFTER_OP` (cast-after-operation)

This example sets parameters for a code replacement operator entry for a cast-after-operation implementation of a `uint8` addition.

```
op_entry = RTW.TflCOperationEntry;
setTflCOperationEntryParameters(op_entry, ...
                  'Key',                      'RTW_OP_ADD', ...
                  'EntryInfoAlgorithm',       'RTW_CAST_AFTER_OP', ...
                  'ImplementationName',       'u8_add_u8_u8');
```

For more information, see `setTflCOperationEntryParameters`.

Algorithm Classification

During code generation, the code generator examines addition and subtraction operations, including adjacent type cast operations, to determine the type of algorithm to compute the expression result. Based on the data types in the expression and the type of the accumulator (type used to hold the result of the addition or subtraction operation), the code generator uses these rules.

- Floating-point types only

Input 1 Data Type	Input 2 Data Type	Accumulator Data Type	Output Data Type	Classification
`double`	`double`	`double`	`double`	CBO, CAO
`double`	`double`	`double`	`single`	—
`double`	`double`	`single`	`double`	—
`double`	`double`	`single`	`single`	CBO
`double`	`single`	`double`	`double`	CBO, CAO

Input 1 Data Type	Input 2 Data Type	Accumulator Data Type	Output Data Type	Classification
`double`	`single`	`double`	`single`	—
`double`	`single`	`single`	`double`	—
`double`	`single`	`single`	`single`	CBO
`single`	`single`	`single`	`single`	CBO, CAO
`single`	`single`	`single`	`double`	—
`single`	`single`	`double`	`single`	—
`single`	`single`	`double`	`double`	CBO, CAO

- Floating-point and fixed-point types on the immediate addition or subtraction operation

Algorithm	Conditions
CBO	One of the following is true: • Operation type is `double`. • Operation type is `single` and input types are `single` or fixed-point.
CAO	Operation type is a superset of input types—that is, output type can represent values of input types without loss of data.

- Fixed-point types only

Algorithm	Conditions
CBO	At least one of the following is true: • Accumulator type equals output type (`Tacc == Tout`). • Output type is a superset of input types (`Tacc >= {Tin1, Tin2}`) and accumulator type is a superset of output type (`Tacc >= Tout`). • Operation does not incur range or precision loss.
CAO	Net bias is zero and the data types in the expression have equal slope adjustment factors. For more information on net bias, see "Addition" or "Subtraction" in "Fixed-Point Operator Code Replacement" on page 52-146 (for MATLAB code) or "Fixed-Point Operator Code Replacement" on page 51-195 (for Simulink models).

In many cases, the numerical result of a CBO operation is equal to that of a CAO operation. For example, if the input and output types are such that the operation produces the ideal result, as in the case of `int8 + int8 -> int16`. To maximize the probability of code replacement occurring in such cases, set the algorithm to cast-after-operation.

Limitations

- The code generator does not replace operations with nonzero net bias.
- When classifying an operation as a CAO operation, the code generator includes the adjacent casts in the expression when the expression involves only fixed-point types. Otherwise, the code generator classifies and replaces only the immediate addition or subtraction operation. Casts that the code generator excludes from the classification appear in the generated code.
- To enable the code generator to include multiple cast operations, which follow an addition or subtraction of fixed-point data, in the classification of an expression, the rounding mode must be `simplest` or `floor`. Consider the expression `y=(cast A)(cast B)(u1+u2)`. If the rounding mode of `(cast A)`, `(cast B)`, and the addition operator (+) are set to `simplest` or `floor`, the code generator takes into account `(cast A)` and `(cast B)` when classifying the expression and performing the replacement only.

More About

- "Code You Can Replace from MATLAB Code" on page 52-5
- "Define Code Replacement Mappings" on page 52-30
- "Remap Operator Output to Function Input" on page 52-143
- "Customize Match and Replacement Process" on page 52-112
- "Fixed-Point Operator Code Replacement" on page 52-146
- "Develop a Code Replacement Library" on page 52-15

Small Matrix Operation to Processor Code Replacement

This example shows how to define code replacement mappings that replace nonscalar small matrix operations with processor-specific intrinsic functions. The example defines a table containing two matrix operator replacement entries for the + (addition) operator and the `double` data type. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

1. Create a table definition file that contains a function definition. For example:

   ```
   function hTable = crl_table_matrix_add_double
   ```

2. Within the function body, create the table by calling the function `RTW.TflTable`.

   ```
   hTable = RTW.TflTable;
   ```

3. Create the entry for the first operator mapping with a call to the `RTW.TflCOperationEntry` function.

   ```
   % Create table entry for matrix_sum_2x2_double
   op_entry = RTW.TflCOperationEntry;
   ```

4. Set operator entry parameters with a call to the `setTflCOperationEntryParameters` function. The code generator ignores saturation and rounding modes for floating-point nonscalar addition and subtraction. For code replacement entries for nonscalar addition and subtraction operations that do not involve fixed-point data, in the call to `setTflCOperationEntryParameters`, specify `'RTW_SATURATE_UNSPECIFIED'` for the `SaturationMode` property and `{'RTW_ROUND_UNSPECIFIED'}` for `RoundingModes`.

   ```
   setTflCOperationEntryParameters(op_entry, ...
       'Key',                    'RTW_OP_ADD', ...
       'Priority',               30, ...
       'SaturationMode',         'RTW_WRAP_ON_OVERFLOW', ...
       'ImplementationName',     'matrix_sum_2x2_double', ...
       'ImplementationHeaderFile', 'MatrixMath.h', ...
       'ImplementationSourceFile', 'MatrixMath.c', ...
       'ImplementationHeaderPath', LibPath, ...
       'ImplementationSourcePath', LibPath, ...
       'AdditionalIncludePaths',   {LibPath}, ...
       'GenCallback',            'RTW.copyFileToBuildDir', ...
       'SideEffects',            true);
   ```

5. Create conceptual arguments `y1`, `u1`, and `u2`. There are multiple ways to set up the conceptual arguments. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. To specify a matrix argument in the function call, use the argument

class `RTW.TflArgMatrix`. Specify the base type and the dimensions for which the argument is valid. The first table entry specifies `[2 2]` and the second table entry specifies `[3 3]`.

```
% Specify operands and result
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
                          'Name',         'y1', ...
                          'IOType',       'RTW_IO_OUTPUT', ...
                          'BaseType',     'double', ...
                          'DimRange',     [2 2]);
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix',...
                          'Name',         'u1', ...
                          'BaseType',     'double', ...
                          'DimRange',     [2 2]);
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix',...
                          'Name',         'u2', ...
                          'BaseType',     'double', ...
                          'DimRange',     [2 2]);
```

6 Create the implementation arguments. There are multiple ways to set up the implementation arguments. This example uses calls to the `getTflArgFromString` to create the arguments. The convenience methods `setReturn` and `addArgument` specify whether an argument is a return value or argument and adds the argument to the entry's array of implementation arguments.

```
arg = getTflArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.setReturn(arg);

arg = getTflArgFromString(hTable, 'u1', ['double' '*']);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u2', ['double' '*']);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'y1', ['double' '*']);
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.addArgument(arg);
```

7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

8 Create the entry for the second operator mapping.

```
% Create table entry for matrix_sum_3x3_double
op_entry = RTW.TflCOperationEntry;
setTflCOperationEntryParameters(op_entry, ...
    'Key',                      'RTW_OP_ADD', ...
    'Priority',                 30, ...
    'SaturationMode',           'RTW_WRAP_ON_OVERFLOW', ...
    'ImplementationName',       'matrix_sum_3x3_double', ...
```

```
        'ImplementationHeaderFile', 'MatrixMath.h', ...
        'ImplementationSourceFile', 'MatrixMath.c', ...
        'ImplementationHeaderPath', LibPath, ...
        'ImplementationSourcePath', LibPath, ...
        'AdditionalIncludePaths',   {LibPath}, ...
        'GenCallback',              'RTW.copyFileToBuildDir', ...
        'SideEffects',              true);

    % Specify operands and result
    createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
                              'Name',         'y1', ...
                              'IOType',       'RTW_IO_OUTPUT', ...
                              'BaseType',     'double', ...
                              'DimRange',     [3 3]);
    createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix',...
                              'Name',         'u1', ...
                              'BaseType',     'double', ...
                              'DimRange',     [3 3]);
    createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix',...
                              'Name',         'u2', ...
                              'BaseType',     'double', ...
                              'DimRange',     [3 3]);

    % Specify replacement function signature
    arg = getTflArgFromString(hTable, 'y2', 'void');
    arg.IOType = 'RTW_IO_OUTPUT';
    op_entry.Implementation.setReturn(arg);
    arg = getTflArgFromString(hTable, 'u1', ['double' '*']);
    op_entry.Implementation.addArgument(arg);
    arg = getTflArgFromString(hTable, 'u2', ['double' '*']);
    op_entry.Implementation.addArgument(arg);
    arg = getTflArgFromString(hTable, 'y1', ['double' '*']);
    arg.IOType = 'RTW_IO_OUTPUT';
    op_entry.Implementation.addArgument(arg);

    addEntry(hTable, op_entry);
```

9 Save the table definition file. Use the name of the table definition function to name the file.

More About

- “Code You Can Replace from MATLAB Code” on page 52-5
- “Define Code Replacement Mappings” on page 52-30

- "Matrix Multiplication Operation to MathWorks BLAS Code Replacement" on page 52-130
- "Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement" on page 52-137
- "Remap Operator Output to Function Input" on page 52-143
- "Customize Match and Replacement Process" on page 52-112
- "Develop a Code Replacement Library" on page 52-15

Matrix Multiplication Operation to MathWorks BLAS Code Replacement

This example shows how to define code replacement mappings that replace nonscalar multiplication operations with Basic Linear Algebra Subroutine (BLAS) multiplication functions `xgemm` and `xgemv`. The example defines code replacement entries that map floating-point matrix/matrix and matrix/vector multiplication operations to MathWorks BLAS library multiplication functions `dgemm` and `dgemv`. The example defines the function mappings programmatically. Alternatively, you can use the Code Replacement Tool to define the same mappings.

BLAS libraries support matrix/matrix multiplication in the form of $C = a(op(A) * op(B)) + bC$. $op(X)$ means X, transposition of X, or Hermitian transposition of X. However, code replacement libraries support only the limited case of $C = op(A) * op(B)\ (a = 1.0, b = 0.0)$. Correspondingly, although BLAS libraries support matrix/vector multiplication in the form of $y = a(op(A) * x) + by$, code replacement libraries support only the limited case of $y = op(A) * x\ (a = 1.0, b = 0.0)$.

1. Create a table definition file that contains a function definition. For example:

   ```
   function hTable = crl_table_tmwblas_mmult_double
   ```

2. Within the function body, create the table by calling the function `RTW.TflTable`.

   ```
   hTable = RTW.TflTable;
   ```

3. Define the path for the BLAS function library. If your replacement functions are on the MATLAB search path or are in your working folder, you can skip this step.

   ```
   % Define library path for Windows or UNIX
   arch = computer('arch');
   if ~ispc
       LibPath = fullfile('$(MATLAB_ROOT)', 'bin', arch);
   else
       % Use Stateflow to get the compiler info
       compilerInfo = sf('Private','compilerman','get_compiler_info');
       compilerName = compilerInfo.compilerName;
       if strcmp(compilerName, 'msvc90') || ...
               strcmp(compilerName, 'msvc80') || ...
               strcmp(compilerName, 'msvc71') || ...
               strcmp(compilerName, 'msvc60'), ...
               compilerName = 'microsoft';
       end
       LibPath = fullfile('$(MATLAB_ROOT)', 'extern', 'lib', arch, compilerName);
   end
   ```

4 Create an entry for the first mapping with a call to the `RTW.TflBlasEntryGenerator` function.

```
% Create table entry for dgemm32
op_entry = RTW.TflBlasEntryGenerator;
```

5 Set operator entry parameters with a call to the `setTflCFunctionEntryParameters` function. The function call sets matrix multiplication operator entry properties. The code generator ignores saturation and rounding modes for floating-point nonscalar addition and subtraction. For code replacement entries for nonscalar addition and subtraction operations that do not involve fixed-point data, in the call to `setTflCFunctionEntryParameters`, specify `'RTW_SATURATE_UNSPECIFIED'` for the `SaturationMode` property and `{'RTW_ROUND_UNSPECIFIED'}` for `RoundingModes`.

```
if ispc
    libExt = 'lib';
elseif ismac
    libExt = 'dylib';
else
    libExt = 'so';
end
setTflCOperationEntryParameters(op_entry, ...
    'Key',                      'RTW_OP_MUL', ...
    'Priority',                 100, ...
    'ImplementationName',       'dgemm32', ...
    'ImplementationHeaderFile', 'blascompat32_crl.h', ...
    'ImplementationHeaderPath', fullfile('$(MATLAB_ROOT)','extern','include'), ...
    'AdditionalLinkObjs',       {['libmwblascompat32.' libExt]}, ...
    'AdditionalLinkObjsPaths',  {LibPath}, ...
    'SideEffects',              true);
```

6 Create conceptual arguments `y1`, `u1`, and `u2`. There are multiple ways to set up the conceptual arguments. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. To specify a matrix argument in the function call, use the argument class `RTW.TflArgMatrix` and specify the base type and the dimensions for which the argument is valid. This type of table entry supports a range of dimensions specified in the format `[Dim1Min Dim2Min ... DimNMin; Dim1Max Dim2Max ... DimNMax]`. For example, `[2 2; inf inf]` means a two-dimensional matrix of size 2x2 or larger. The conceptual output argument for the `dgemm32` entry for matrix/matrix multiplication replacement specifies dimensions `[2 2; inf inf]`, while the conceptual output argument for the `dgemv32` entry for matrix/vector multiplication replacement specifies dimensions `[2 1; inf 1]`.

```
% Specify operands and result
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
                          'Name',         'y1', ...
```

```
                            'IOType',       'RTW_IO_OUTPUT', ...
                            'BaseType',     'double', ...
                            'DimRange',     [2 2; inf inf]);
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
                            'Name',         'u1', ...
                            'BaseType',     'double', ...
                            'DimRange',     [2 2; inf inf]);
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
                            'Name',         'u2', ...
                            'BaseType',     'double', ...
                            'DimRange',     [1 1; inf inf]);
```

7 Create the implementation arguments. There are multiple ways to set up the implementation arguments. This example uses calls to the `getTflArgFromString` and `RTW.TflArgCharConstant` functions to create the arguments. The example code configures special implementation arguments that are required for `dgemm` and `dgemv` function replacements. The convenience methods `setReturn` and `addArgument` specify whether an argument is a return value or argument and adds the argument to the entry's array of implementation arguments.

```
% Using RTW.TflBlasEntryGenerator for xgemm requires the following
% implementation signature:
%
% void f(char* TRANSA, char* TRANSB, int* M, int* N, int* K,
%        type* ALPHA, type* u1, int* LDA, type* u2, int* LDB,
%        type* BETA, type* y, int* LDC)
%
% When a match occurs, the code generator computes the
% values for M, N, K, LDA, LDB, and LDC and inserts them into the
% generated code. TRANSA and TRANSB are set to 'N'.

% Specify replacement function signature

arg = getTflArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.setReturn(arg);

arg = RTW.TflArgCharConstant('TRANSA');
% Possible values for PassByType property are
%  RTW_PASSBY_AUTO, RTW_PASSBY_POINTER,
%  RTW_PASSBY_VOID_POINTER, RTW_PASSBY_BASE_POINTER
arg.PassByType = 'RTW_PASSBY_POINTER';
op_entry.Implementation.addArgument(arg);

arg = RTW.TflArgCharConstant('TRANSB');
arg.PassByType = 'RTW_PASSBY_POINTER';
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'M', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);
```

```
arg = getTflArgFromString(hTable, 'N', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'K', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'ALPHA', 'double', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u1', ['double' '*']);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'LDA', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u2', ['double' '*']);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'LDB', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'BETA', 'double', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'y1', ['double' '*']);
arg.IOType = 'RTW_IO_OUTPUT';
arg.PassByType = 'RTW_PASSBY_POINTER';
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'LDC', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);
```

8 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

9 Create the entry for the second mapping.

```
% Create table entry for dgemv32
op_entry = RTW.TflBlasEntryGenerator;
if ispc
    libExt = 'lib';
elseif ismac
    libExt = 'dylib';
else
    libExt = 'so';
end
setTflCOperationEntryParameters(op_entry, ...
    'Key',                    'RTW_OP_MUL', ...
    'Priority',               100, ...
    'ImplementationName',     'dgemv32', ...
    'ImplementationHeaderFile', 'blascompat32_crl.h', ...
    'ImplementationHeaderPath', fullfile('$(MATLAB_ROOT)','extern','include'), ...
    'AdditionalLinkObjs',     {['libmwblascompat32.' libExt]}, ...
    'AdditionalLinkObjsPaths',  {LibPath},...
    'SideEffects',            true);

% Specify operands and result
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
                          'Name',         'y1', ...
                          'IOType',       'RTW_IO_OUTPUT', ...
                          'BaseType',     'double', ...
                          'DimRange',     [2 1; inf 1]);
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
                          'Name',         'u1', ...
                          'BaseType',     'double', ...
                          'DimRange',     [2 2; inf inf]);
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix',...
                          'Name',         'u2', ...
                          'BaseType',     'double', ...
                          'DimRange',     [1 1; inf 1]);

% Using RTW.TflBlasEntryGenerator for xgemv requires the following
% implementation signature:
%
% void f(char* TRANS, int* M, int* N,
%        type* ALPHA, type* u1, int* LDA, type* u2, int* INCX,
%        type* BETA, type* y, int* INCY)
%
% Upon a match, the CRL entry will compute the
% values for M, N, LDA, INCX, and INCY, and insert them into the
% generated code. TRANS will be set to 'N'.

% Specify replacement function signature

arg = getTflArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.setReturn(arg);

arg = RTW.TflArgCharConstant('TRANS');
arg.PassByType = 'RTW_PASSBY_POINTER';
op_entry.Implementation.addArgument(arg);
```

```
arg = getTflArgFromString(hTable, 'M', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'N', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'ALPHA', 'double', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u1', ['double' '*']);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'LDA', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u2', ['double' '*']);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'INCX','integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'BETA', 'double', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'y1', ['double' '*']);
arg.IOType = 'RTW_IO_OUTPUT';
arg.PassByType = 'RTW_PASSBY_POINTER';
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'INCY', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

addEntry(hTable, op_entry);
```

10 Save the table definition file. Use the name of the table definition function to name the file.

More About

- "Code You Can Replace from MATLAB Code" on page 52-5
- "Define Code Replacement Mappings" on page 52-30
- "Small Matrix Operation to Processor Code Replacement" on page 52-126
- "Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement" on page 52-137
- "Remap Operator Output to Function Input" on page 52-143
- "Customize Match and Replacement Process" on page 52-112
- "Develop a Code Replacement Library" on page 52-15

Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement

This example shows how to define code replacement mappings that replace nonscalar multiplication operations with ANSI/ISO C BLAS multiplication functions `xgemm` and `xgemv`. The example defines code replacement entries that map floating-point matrix/matrix and matrix/vector multiplication operations to ANSI/ISO C BLAS library multiplication functions `dgemm` and `dgemv`. The example defines the function mappings programmatically. Alternatively, you can use the Code Replacement Tool to define the same mappings.

BLAS libraries support matrix/matrix multiplication in the form of $C = a(op(A) * op(B)) + bC$. $op(X)$ means X, transposition of X, or Hermitian transposition of X. However, code replacement libraries support only the limited case of $C = op(A) * op(B)\ (a = 1.0, b = 0.0)$. Correspondingly, although BLAS libraries support matrix/vector multiplication in the form of $y = a(op(A) * x) + by$, code replacement libraries support only the limited case of $y = op(A) * x\ (a = 1.0, b = 0.0)$.

1. Create a table definition file that contains a function definition. For example:

   ```
   function hTable = crl_table_cblas_mmult_double
   ```

2. Within the function body, create the table by calling the function `RTW.TflTable`.

   ```
   hTable = RTW.TflTable;
   ```

3. Define the path for the CBLAS function library. For example:

   ```
   LibPath = fullfile(matlabroot, 'toolbox', 'rtw', 'rtwdemos', 'crl_demo');
   ```

4. Create an entry for the first mapping with a call to the `RTW.TflBlasEntryGenerator` function.

   ```
   % Create table entry for cblas_dgemm
   op_entry = RTW.TflCBlasEntryGenerator;
   ```

5. Set operator entry parameters with a call to the `setTflCOperationEntryParameters` function. The function call sets matrix multiplication operator entry properties. The code generator ignores saturation and rounding modes for floating-point nonscalar addition and subtraction.

   ```
   setTflCOperationEntryParameters(op_entry, ...
       'Key',                      'RTW_OP_MUL', ...
   ```

```
    'Priority',                100, ...
    'ImplementationName',      'cblas_dgemm', ...
    'ImplementationHeaderFile', 'cblas.h', ...
    'ImplementationHeaderPath', LibPath, ...
    'AdditionalIncludePaths',   {LibPath}, ...
    'GenCallback',             'RTW.copyFileToBuildDir', ...
    'SideEffects',             true);
```

6 Create conceptual arguments `y1`, `u1`, and `u2`. There are multiple ways to set up the conceptual arguments. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. To specify a matrix argument in the function call, use the argument class `RTW.TflArgMatrix` and specify the base type and the dimensions for which the argument is valid. This type of table entry supports a range of dimensions specified in the format `[Dim1Min Dim2Min ... DimNMin; Dim1Max Dim2Max ... DimNMax]`. For example, `[2 2; inf inf]` means a two-dimensional matrix of size 2x2 or larger. The conceptual output argument for the `dgemm32` entry for matrix/matrix multiplication replacement specifies dimensions `[2 2; inf inf]`. The conceptual output argument for the `dgemv32` entry for matrix/vector multiplication replacement specifies dimensions `[2 1; inf 1]`.

```
% Specify operands and result
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
                          'Name',         'y1', ...
                          'IOType',       'RTW_IO_OUTPUT', ...
                          'BaseType',     'double', ...
                          'DimRange',     [2 2; inf inf]);
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
                          'Name',         'u1', ...
                          'BaseType',     'double', ...
                          'DimRange',     [2 2; inf inf]);
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
                          'Name',         'u2', ...
                          'BaseType',     'double', ...
                          'DimRange',     [1 1; inf inf]);
```

7 Create the implementation arguments. There are multiple ways to set up the implementation arguments. This example uses calls to the `getTflArgFromString` function to create the arguments. The example code configures special implementation arguments that are required for `dgemm` and `dgemv` function replacements. The convenience methods `setReturn` and `addArgument` specify whether an argument is a return value or argument and adds the argument to the entry's array of implementation arguments.

```
% Using RTW.TflCBlasEntryGenerator for xgemm requires the following
% implementation signature:
%
% void f(enum ORDER, enum TRANSA, enum TRANSB, int M, int N, int K,
%        type ALPHA, type* u1, int LDA, type* u2, int LDB,
```

```
%        type BETA, type* y, int LDC)
%
% Since CRLs do not have the ability to specify enums, you must
% use integer.  (This will cause problems with C++ code generation,
% so for C++, use a wrapper function to cast each int to the
% corresponding enumeration type.)
%
% When a match occurs, the code generator computes the
% values for M, N, K, LDA, LDB, and LDC and insert them into the
% generated code.

% Specify replacement function signature

arg = getTflArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.setReturn(arg);

arg = getTflArgFromString(hTable, 'ORDER', 'integer', 102);
% arg.Type.ReadOnly=true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'TRANSA', 'integer', 111);
% arg.Type.ReadOnly=true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'TRANSB', 'integer', 111);
% arg.Type.ReadOnly=true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'M', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'N', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'K', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'ALPHA', 'double', 1);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u1', ['double' '*']);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'LDA', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u2', ['double' '*']);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'LDB', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'BETA', 'double', 0);
op_entry.Implementation.addArgument(arg);
```

```
arg = getTflArgFromString(hTable, 'y1', ['double' '*']);
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'LDC', 'integer', 0);
op_entry.Implementation.addArgument(arg);
```

8 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

9 Create the entry for the second mapping.

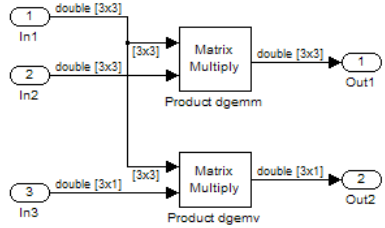
```
% Create table entry for cblas_dgemv
op_entry = RTW.TflCBlasEntryGenerator;
setTflCOperationEntryParameters(op_entry, ...
    'Key',                      'RTW_OP_MUL', ...
    'Priority',                 100, ...
    'ImplementationName',       'cblas_dgemv', ...
    'ImplementationHeaderFile', 'cblas.h', ...
    'ImplementationHeaderPath', LibPath, ...
    'AdditionalIncludePaths',   {LibPath}, ...
    'GenCallback',              'RTW.copyFileToBuildDir', ...
    'SideEffects',              true);

% Specify operands and result
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
                          'Name',         'y1', ...
                          'IOType',       'RTW_IO_OUTPUT', ...
                          'BaseType',     'double', ...
                          'DimRange',     [2 1; inf 1]);
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
                          'Name',         'u1', ...
                          'BaseType',     'double', ...
                          'DimRange',     [2 2; inf inf]);
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix',...
                          'Name',         'u2', ...
                          'BaseType',     'double', ...
                          'DimRange',     [1 1; inf 1]);

% Using RTW.TflCBlasEntryGenerator for xgemv requires the following
% implementation signature:
%
% void f(enum ORDER, enum TRANSA, int M, int N,
%        type ALPHA, type* u1, int LDA, type* u2, int INCX,
%        type BETA, type* y, int INCY)
%
% Since CRLs do not have the ability to specify enums, you must
% use integer.  (This will cause problems with C++ code generation,
% so for C++, use a wrapper function to cast each int to the
% corresponding enumeration type.)
%
% Upon a match, the CRL entry will compute the
% values for M, N, LDA, INCX, and INCY and insert them into the
% generated code.
```

```
% Specify replacement function signature

arg = getTflArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.setReturn(arg);

arg = getTflArgFromString(hTable, 'ORDER', 'integer', 102);
% arg.Type.ReadOnly=true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'TRANSA', 'integer', 111);
% arg.Type.ReadOnly=true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'M','integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'N', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'ALPHA', 'double', 1);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u1', ['double' '*']);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'LDA', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u2', ['double' '*']);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'INCX', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'BETA', 'double', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'y1', ['double' '*']);
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'INCY', 'integer', 0);
op_entry.Implementation.addArgument(arg);

addEntry(hTable, op_entry);
```

10 Save the table definition file. Use the name of the table definition function to name the file.

To test this example, create a model that uses two Product blocks. For example:

1 Create a model that includes two Product blocks, such as the following:



2 Configure the model with the following settings:

- On the **Solver** pane, select a fixed-step, discrete solver with a fixed-step size such as `0.1`.
- On the **Code Generation** pane, select an ERT-based system target file.
- On the **Code Generation** > **Interface** pane, select the code replacement library that contains your addition operation entry.

3 For each Product block, set the block parameter **Multiplication** to the value `Matrix(*)`.

4 In the Model Explorer, configure the **Signal Attributes** for the `In1`, `In2`, and `In3` source blocks. For `In1` and `In2`, set **Port dimensions** to `[3 3]` and set the **Data type** to `double`. For `In3`, set **Port dimensions** to `[3 1]` and set the **Data type** to `double`.

5 Generate code and a code generation report.

6 Review the code replacements.

More About

- "Code You Can Replace from MATLAB Code" on page 52-5
- "Define Code Replacement Mappings" on page 52-30
- "Small Matrix Operation to Processor Code Replacement" on page 52-126
- "Matrix Multiplication Operation to MathWorks BLAS Code Replacement" on page 52-130
- "Remap Operator Output to Function Input" on page 52-143
- "Customize Match and Replacement Process" on page 52-112
- "Develop a Code Replacement Library" on page 52-15

Remap Operator Output to Function Input

If your generated code must meet a specific coding pattern or you want more flexibility, for example, to further improve performance, you can remap operator outputs to input positions in an implementation function argument list.

Note: Remapping outputs to implementation function inputs is supported only for operator replacement.

For example, for a sum operation, the code generator produces code similar to:

```
add8_Y.Out1 = u8_add_u8_u8(add8_U.In1, add8_U.In2);
```

If you remap the output to the first input, the code generator produces code similar to:

```
u8_add_u8_u8(&add8_Y.Out1;, add8_U.In1, add8_U.In2);
```

The following table definition file for a sum operation remaps operator output `y1` as the first function input argument.

1. Create a table definition file that contains a function definition. For example:

   ```
   function hTable = crl_table_add_uint8
   ```

2. Within the function body, create the table by calling the function `RTW.TflTable`.

   ```
   hTable = RTW.TflTable;
   ```

3. Create an entry for the operator mapping with a call to the `RTW.TflCOperationEntry` function.

   ```
   % Create operation entry
   op_entry = RTW.TflCOperationEntry;
   ```

4. Set operator entry parameters with a call to the `setTflCOperationEntryParameters` function. In the function call, set the property `SideEffects` to `true`.

   ```
   setTflCOperationEntryParameters(op_entry, ...
                       'Key',                      'RTW_OP_ADD', ...
                       'Priority',                 90, ...
                       'ImplementationName',       'u8_add_u8_u8', ...
                       'ImplementationHeaderFile', 'u8_add_u8_u8.h', ...
                       'ImplementationSourceFile', 'u8_add_u8_u8.c', ...
                       'SideEffects',              true );
   ```

5 Create conceptual arguments `y1`, `u1`, and `u2`. There are multiple ways to set up the conceptual arguments. This example uses calls to the `getTflArgFromString` and `addConceptualArg` functions to create and add the arguments.

```
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 );
```

6 Create the implementation arguments. There are multiple ways to set up the implementation arguments. This example uses calls to the `getTflArgFromString` function to create the arguments. When defining the implementation function return argument, create a new `void` output argument, for example, `y2`. When defining the implementation function argument for the conceptual output argument (`y1`), set the operator output argument as an additional input argument. Mark its `IOType` as output. Make its type a pointer type. The convenience methods `setReturn` and `addArgument` specify whether an argument is a return value or argument and adds the argument to the entry's array of implementation arguments.

```
% Create new void output y2
arg = getTflArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.setReturn(arg);

% Set y1 as first input arg, mark IOType as output, and use pointer type
arg=getTflArgFromString(hTable, 'y1', 'uint8*');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.addArgument(arg);

arg=getTflArgFromString(hTable, 'u1', 'uint8');
op_entry.Implementation.addArgument(arg);

arg=getTflArgFromString(hTable, 'u2', 'uint8');
op_entry.Implementation.addArgument(arg);
```

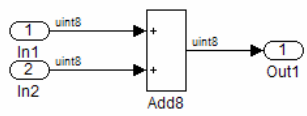
7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

8 Save the table definition file. Use the name of the table definition function to name the file.

To test this example, create a model that uses an Add block. For example:

1 Create a model that includes an Add block, such as the following:



2 Configure the model with the following settings:

- On the **Solver** pane, select a fixed-step solver.
- On the **Code Generation** pane, select an ERT-based system target file.
- On the **Code Generation** > **Interface** pane, select the code replacement library that contains your addition operation entry.
- On the **All Parameters** tab, set the **Optimize global data access** parameter to `Use global to hold temporary results`. This reduces data copies in the generated code.

3 Generate code and a code generation report.

4 Review the code replacements.

More About

- “Code You Can Replace from MATLAB Code” on page 52-5
- “Define Code Replacement Mappings” on page 52-30
- “Develop a Code Replacement Library” on page 52-15

Fixed-Point Operator Code Replacement

If you have a Fixed-Point Designer license, you can define fixed-point operator code replacement entries to match:

- A binary-point-only scaling combination on the operator inputs and output.
- A slope bias scaling combination on the operator inputs and output.
- Relative scaling or net slope between multiplication or division operator inputs and output. Use one of these methods to map a range of slope and bias values to a replacement function for multiplication or division.
- Equal slope and zero net bias across addition or subtraction operator inputs and output. Use this method to disregard specific slope and bias values and map relative slope and bias values to a replacement function for addition or subtraction.

Common Ways to Match Fixed-Point Operator Entries

The following table maps common ways to match fixed-point operator code replacement entries with the associated fixed-point parameters that you specify in a code replacement table definition file.

Match	Create entry	Minimally specify parameters
A specific binary-point-only scaling combination on the operator inputs and output.	`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.	`RTW.TflCOperationEntry`	`createAndAddConceptualArg` function:

Match	Create entry	Minimally specify parameters
		• `CheckSlope`: Specify the value `true`. • `CheckBias`: Specify the value `true`. • `DataTypeMode` (or `DataType/Scaling` equivalent): Specify fixed-point [slope bias] scaling. • `Slope` (or `SlopeAdjustmentFactor/FixedExponent` equivalent): Specify a slope value (for example, `15`). • `Bias`: Specify a bias value (for example, `2`).
Net slope between operator inputs and output (multiplication and division).	`RTW.TflCOperationEntryGenerator_NetSlope`	`setTflCOperationEntryParameters` function: • `NetSlopeAdjustmentFactor`: Specify the slope adjustment factor (`F`) part of the net slope, $F2^E$ (for example, `1.0`). • `NetFixedExponent`: Specify the fixed exponent (`E`) part of the net slope, $F2^E$ (for example, `-3.0`). `createAndAddConceptualArg` function: • `CheckSlope`: Specify the value `false`. • `CheckBias`: Specify the value `false`. • `DataType`: Specify the value `'Fixed'`.

Match	Create entry	Minimally specify parameters
Relative scaling between operator inputs and output (multiplication and division).	`RTW.TflCOperationEntryGenerator`	`setTflCOperationEntryParameters` function: • `RelativeScalingFactorF`: Specify the slope adjustment factor (`F`) part of the relative scaling factor, $F2^E$ (for example, `1.0`). • `RelativeScalingFactorE`: Specify the fixed exponent (`E`) part of the relative scaling factor, $F2^E$ (for example, `-3.0`). `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).	`RTW.TflCOperationEntryGenerator`	`setTflCOperationEntryParameters` function: • `SlopesMustBeTheSame`: Specify the value `true`. • `MustHaveZeroNetBias`: Specify the value `true`. `createAndAddConceptualArg` function: • `CheckSlope`: Specify the value `false`. • `CheckBias`: Specify the value `false`.

Fixed-Point Numbers and Arithmetic

Fixed-point numbers use integers and integer arithmetic to represent real numbers and arithmetic with the following encoding scheme:

$$V = \tilde{V} = SQ + B$$

- 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:

$$\left(S_O Q_O + B_O\right) = \left(S_1 Q_1 + B_1\right) < op > (S_2 Q_2 + B_2)$$

The objective of 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. The following sections provide additional programming information for each supported operator.

Addition

The operation $V_0 = V_1 + V_2$ implies that

$$Q_0 = \left(\frac{S_1}{S_0}\right) Q_1 + \left(\frac{S_2}{S_0}\right) Q_2 + \left(\frac{B_1 + B_2 - B_0}{S_0}\right)$$

If an addition replacement function is defined such that the scaling on the operands and sum are equal and the net bias

$$\left(\frac{B_1 + B_2 - B_0}{S_0} \right)$$

is zero (for example, a function `s8_add_s8_s8` that adds two signed 8-bit values and produces a signed 8-bit result), then the 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 $V_0 = V_1 - V_2$ implies that

$$Q_0 = \left(\frac{S_1}{S_0} \right) Q_1 - \left(\frac{S_2}{S_0} \right) Q_2 + \left(\frac{B_1 - B_2 - B_0}{S_0} \right)$$

If a subtraction replacement function is defined such that the scaling on the operands and difference are equal and the net bias

$$\left(\frac{B_1 - B_2 - B_0}{S_0} \right)$$

is zero (for example, a function `s8_sub_s8_s8` that subtracts two signed 8-bit values and produces a signed 8-bit result), then the 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

There are different ways to specify multiplication replacements. The most direct way is to specify an exact match of the input and output types. This is feasible if a model contains only a few (known) slope and bias combinations. Use the `TflCOperationEntry` class

and specify the exact values of slope and bias on each argument. For scenarios where there are numerous slope/bias combinations, it is not feasible to specify each value with a different entry. Use a net slope entry or create a custom entry.

The operation $V_0 = V_1 * V_2$ implies, for binary-point-only scaling, that

$$S_0 Q_0 = (S_1 Q_1)(S_2 Q_2)$$
$$Q_0 = \left(\frac{S_1 S_2}{S_0} \right) Q_1 Q_2$$
$$Q_0 = S_n Q_1 Q_2$$

where S_n is the net slope.

It is common to replace all multiplication operations that have a net slope of 1.0 with a function that performs C-style multiplication. For example, to replace all signed 8-bit multiplications that have a net scaling of 1.0 with the `s8_mul_s8_u8_` replacement function, the operator entry must define a net slope factor, $F2^E$. You specify the values for F and E using operator entry parameters `NetSlopeAdjustmentFactor` and `NetFixedExponent`. (For parameter descriptions, see the reference page for the function `setTflCOperationEntryParameters`.) For the `s8_mul_s8_u8` function, set `NetSlopeAdjustmentFactor` to 1 and `NetFixedExponent` to 0.0.

Note: When an operator entry specifies `NetSlopeAdjustmentFactor` and `NetFixedExponent`, matching entries must have arguments with zero bias.

Division

There are different ways to specify division replacements. The most direct way is to specify an exact match of the input and output types. This is feasible if a model contains only a few (known) slope and bias combinations. For this, use the `TflCOperationEntry` class and specify the exact values of slope and bias on each argument. For scenarios where there are numerous slope/bias combinations, it is not feasible to specify each value with a different entry. For this, use a net slope entry or create a custom entry (see "Customize Match and Replacement Process" on page 51-153).

The operation $V_0 = (V_1 / V_2)$ implies, for binary-point-only scaling, that

$$S_0 Q_0 = \left(\frac{S_1 Q_1}{S_2 Q_2} \right)$$

$$Q_0 = S_n \left(\frac{Q_1}{Q_2} \right)$$

where S_n is the net slope.

It is common to replace all division operations that have a net slope of 1.0 with a function that performs C-style division. For example, to replace all signed 8-bit divisions that have a net scaling of 1.0 with the `s8_mul_s8_u8_` replacement function, the operator entry must define a net slope factor, $F2^E$. You specify the values for F and E using operator entry parameters `NetSlopeAdjustmentFactor` and `NetFixedExponent`. (For parameter descriptions, see the reference page for the function `setTflCOperationEntryParameters`.) For the `s16_netslopeOp5_div_s16_s16` function, you would set `NetSlopeAdjustmentFactor` to 1 and `NetFixedExponent` to 0.0.

Note: When an operator entry specifies `NetSlopeAdjustmentFactor` and `NetFixedExponent`, matching entries must have arguments with zero bias.

Data Type Conversion (Cast)

The data type conversion operation $V_0 = V_1$ implies, for binary-point-only scaling, that

$$Q_0 = \left(\frac{S_1}{S_0} \right) Q_1$$

$$Q_0 = S_n Q_1$$

where S_n is the net slope.

Shift

The shift left or shift right operation $V_0 = (V_1 / 2^n)$ implies, for binary-point-only scaling, that

$$S_0 Q_0 = \left(\frac{S_1 Q_1}{2^n} \right)$$

$$Q_0 = \left(\frac{S_1}{S_0} \right) + \left(\frac{Q_1}{2^n} \right)$$

$$Q_0 = S_n \left(\frac{Q_1}{2^n} \right)$$

where S_n is the net slope.

More About

- "Code You Can Replace from MATLAB Code" on page 52-5
- "Define Code Replacement Mappings" on page 52-30
- "Binary-Point-Only Scaling Code Replacement" on page 52-154
- "Slope Bias Scaling Code Replacement" on page 52-157
- "Net Slope Scaling Code Replacement" on page 52-160
- "Equal Slope and Zero Net Bias Code Replacement" on page 52-166
- "Data Type Conversions (Casts) and Operator Code Replacement" on page 52-169
- "Shift Left Operations and Code Replacement" on page 52-173
- "Remap Operator Output to Function Input" on page 52-143
- "Customize Match and Replacement Process" on page 52-112
- "Develop a Code Replacement Library" on page 52-15

Binary-Point-Only Scaling Code Replacement

You can define code replacement entries for operations on fixed-point data types such that they match a binary-point-only scaling combination on 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 creates a code replacement entry for multiplication of fixed-point data types. You specify arguments using binary-point-only scaling. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

1. Create a table definition file that contains a function definition. For example:

   ```
   function hTable = crl_table_fixed_binptscale
   ```

2. Within the function body, create the table by calling the function `RTW.TflTable`.

   ```
   hTable = RTW.TflTable;
   ```

3. Create the entry for the operator mapping with a call to the `RTW.TflCOperationEntry` function.

   ```
   op_entry = RTW.TflCOperationEntry;
   ```

4. Set operator entry parameters with a call to the `setTflCOperationEntryParameters` function. The parameters specify the type of operation as multiplication, the saturation mode as saturate on integer overflow, rounding modes as unspecified, and the name of the replacement function as `s32_mul_s16_s16_binarypoint`.

   ```
   setTflCOperationEntryParameters(op_entry, ...
       'Key',                    'RTW_OP_MUL', ...
       'Priority',               90, ...
       'SaturationMode',         'RTW_SATURATE_ON_OVERFLOW', ...
       'RoundingModes',          {'RTW_ROUND_UNSPECIFIED'}, ...
       'ImplementationName',     's32_mul_s16_s16_binarypoint', ...
       'ImplementationHeaderFile', 's32_mul_s16_s16_binarypoint.h', ...
       'ImplementationSourceFile', 's32_mul_s16_s16_binarypoint.c');
   ```

5. Create conceptual arguments `y1`, `u1`, and `u2`. There are multiple ways to set up the conceptual arguments. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. 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. The input arguments are 16 bits, signed, with fraction lengths of 15 and 13.

```
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);
```

6 Create the implementation arguments. There are multiple ways to set up the implementation arguments. This example uses calls to the `createAndSetCImplementationReturn` and `createAndAddImplementationArg` functions to create and add implementation arguments to the 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`). The input arguments are 16 bits and signed (`int16`).

```
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);
```

7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

8 Save the table definition file. Use the name of the table definition function to name the file.

More About

- "Code You Can Replace from MATLAB Code" on page 52-5
- "Define Code Replacement Mappings" on page 52-30
- "Fixed-Point Operator Code Replacement" on page 52-146
- "Data Type Conversions (Casts) and Operator Code Replacement" on page 52-169
- "Shift Left Operations and Code Replacement" on page 52-173
- "Remap Operator Output to Function Input" on page 52-143
- "Customize Match and Replacement Process" on page 52-112
- "Develop a Code Replacement Library" on page 52-15

Slope Bias Scaling Code Replacement

You can define code replacement for operations on fixed-point data types as matching a slope bias scaling combination on the operator inputs and output. The slope bias scaling entries can map the specified slope bias combination to a replacement function for addition, subtraction, multiplication, or division.

This example creates a code replacement entry for division of fixed-point data types. You specify arguments using slope bias scaling. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

1. Create a table definition file that contains a function definition. For example:

   ```
   function hTable = crl_table_fixed_s16divslopebias
   ```

2. Within the function body, create the table by calling the function `RTW.TflTable`.

   ```
   hTable = RTW.TflTable;
   ```

3. Create the entry for the operator mapping with a call to the `RTW.TflCOperationEntry` function.

   ```
   op_entry = RTW.TflCOperationEntry;
   ```

4. Set operator entry parameters with a call to the `setTflCOperationEntryParameters` function. The parameters specify the type of operation as division, the saturation mode as saturate on integer overflow, rounding modes as round to ceiling, and the name of the replacement function as `s16_div_s16_s16_slopebias`.

   ```
   setTflCOperationEntryParameters(op_entry, ...
      'Key',                    'RTW_OP_DIV', ...
      'Priority',               90, ...
      'SaturationMode',         'RTW_SATURATE_ON_OVERFLOW', ...
      'RoundingModes',          {'RTW_ROUND_CEILING'}, ...
      'ImplementationName',     's16_div_s16_s16_slopebias', ...
      'ImplementationHeaderFile', 's16_div_s16_s16_slopebias.h', ...
      'ImplementationSourceFile', 's16_div_s16_s16_slopebias.c');
   ```

5. Create conceptual arguments `y1`, `u1`, and `u2`. There are multiple ways to set up the conceptual arguments. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. 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.

```
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);
```

6 Create the implementation arguments. There are multiple ways to set up the implementation arguments. This example uses calls to the `createAndSetCImplementationReturn` and `createAndAddImplementationArg` functions to create and add implementation arguments to the 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`).

```
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);
```

7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

8 Save the table definition file. Use the name of the table definition function to name the file.

More About

- "Code You Can Replace from MATLAB Code" on page 52-5
- "Define Code Replacement Mappings" on page 52-30
- "Fixed-Point Operator Code Replacement" on page 52-146
- "Data Type Conversions (Casts) and Operator Code Replacement" on page 52-169
- "Shift Left Operations and Code Replacement" on page 52-173
- "Remap Operator Output to Function Input" on page 52-143
- "Customize Match and Replacement Process" on page 52-112
- "Develop a Code Replacement Library" on page 52-15

Net Slope Scaling Code Replacement

Multiplication and Division with Saturation

You can define code replacement entries for operations on fixed-point data types as matching net slope between operator inputs and output. The net slope entries can map a range of slope and bias values to a replacement function for multiplication or division.

This example creates a code replacement entry for division of fixed-point data types, using wrap on overflow saturation mode and a net slope. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

1. Create a table definition file that contains a function definition. For example:

   ```
   function hTable = crl_table_fixed_netslopesaturate
   ```

2. Within the function body, create the table by calling the function `RTW.TflTable`.

   ```
   hTable = RTW.TflTable;
   ```

3. Create the entry for the operator mapping with a call to the `RTW.TflCOperationEntryGenerator_Netslope` function, which provides access to the fixed-point parameters `NetSlopeAdjustmentFactor` and `NetFixedExponent`.

   ```
   wv = [16,32];
   for iy = 1:2
     for inum = 1:2
       for iden = 1:2
         hTable = getDivOpEntry(hTable, ...
            fixdt(1,wv(iy)),fixdt(1,wv(inum)),fixdt(1,wv(iden)));
       end
     end
   end


   %------------------------------------------------------------
   function hTable = getDivOpEntry(hTable,dty,dtnum,dtden)
   %------------------------------------------------------------
   % Create an entry for division of fixed-point data types where
   % arguments are specified using Slope and Bias scaling
   % Saturation on, Rounding unspecified

   funcStr = sprintf('user_div_%s_%s_%s',...
           typeStrFunc(dty),...
           typeStrFunc(dtnum),...
           typeStrFunc(dtden));
   ```

```
op_entry = RTW.TflCOperationEntryGenerator_NetSlope;
```

4 Set operator entry parameters with a call to the `setTflCOperationEntryParameters` function. The parameters specify the type of operation as division, the saturation mode as wrap on overflow, rounding modes as unspecified, and the name of the replacement function as `user_div_*`. `NetSlopeAdjustmentFactor` and `NetFixedExponent` specify the `F` and `E` parts of the net slope $F2^E$.

```
setTflCOperationEntryParameters(op_entry, ...
   'Key',                      'RTW_OP_DIV', ...
   'Priority',                 90, ...
   'SaturationMode',           'RTW_WRAP_ON_OVERFLOW',...
   'RoundingModes',            {'RTW_ROUND_UNSPECIFIED'},...
   'NetSlopeAdjustmentFactor', 1.0, ...
   'NetFixedExponent',         0.0, ...
   'ImplementationName',       funcStr, ...
   'ImplementationHeaderFile', [funcStr,'.h'], ...
   'ImplementationSourceFile', [funcStr,'.c']);
```

5 Create conceptual arguments `y1`, `u1`, and `u2`. There are multiple ways to set up the conceptual arguments. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. Specify each argument as fixed-point and signed. Also, for each argument, specify that code replacement request processing does *not* check for an exact match to the call-site slope and bias values.

```
createAndAddConceptualArg(op_entry, ...
   'RTW.TflArgNumeric', ...
   'Name',           'y1',...
   'IOType',         'RTW_IO_OUTPUT',...
   'CheckSlope',     false,...
   'CheckBias',      false,...
   'DataTypeMode',   'Fixed-point: slope and bias scaling',...
   'IsSigned',       dty.Signed,...
   'WordLength',     dty.WordLength,...
   'Bias',           0);

createAndAddConceptualArg(op_entry, ...
   'RTW.TflArgNumeric',...
   'Name',           'u1', ...
   'IOType',         'RTW_IO_INPUT',...
   'CheckSlope',     false,...
   'CheckBias',      false,...
   'DataTypeMode',   'Fixed-point: slope and bias scaling',...
   'IsSigned',       dtnum.Signed,...
   'WordLength',     dtnum.WordLength,...
   'Bias',           0);

createAndAddConceptualArg(op_entry, ...
   'RTW.TflArgNumeric', ...
   'Name',           'u2', ...
```

```
    'IOType',          'RTW_IO_INPUT',...
    'CheckSlope',      false,...
    'CheckBias',       false,...
    'DataTypeMode',    'Fixed-point: slope and bias scaling',...
    'IsSigned',        dtden.Signed,...
    'WordLength',      dtden.WordLength,...
    'Bias',            0);
```

6 Create the implementation arguments. There are multiple ways to set up the implementation arguments. This example uses calls to the `getTflArgFromString` function to create the arguments. Implementation arguments must describe fundamental numeric data types (not fixed-point data types). The convenience methods `setReturn` and `addArgument` specify whether an argument is a return value or argument. These methods add the argument to the entry array of implementation arguments.

```
arg = getTflArgFromString(hTable, 'y1', typeStrBase(dty));
op_entry.Implementation.setReturn(arg);

arg = getTflArgFromString(hTable, 'u1', typeStrBase(dtnum));
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u2',typeStrBase(dtden));
op_entry.Implementation.addArgument(arg);
```

7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

8 Define functions that determine the data type word length.

```
%--------------------------------------------------------------
function str = typeStrFunc(dt)
%--------------------------------------------------------------

if dt.Signed
    sstr = 's';
else
    sstr = 'u';
end
str = sprintf('%s%d',sstr,dt.WordLength);

%--------------------------------------------------------------
function str = typeStrBase(dt)
%--------------------------------------------------------------

if dt.Signed
    sstr = ;
else
    sstr = 'u';
end
str = sprintf('%sint%d',sstr,dt.WordLength);
```

9 Save the table definition file. Use the name of the table definition function to name the file.

Multiplication and Division with Rounding Mode and Additional Implementation Arguments

You can define code replacement entries for multiplication and division operations on fixed-point data types such that they match the net slope between operator inputs and output. The net slope entries can map a range of slope and bias values to a replacement function for multiplication or division.

This example creates a code replacement entry for division of fixed-point data types, using the ceiling rounding mode and a net slope scaling factor. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

1 Create a table definition file that contains a function definition. For example:

```
function hTable = crl_table_fixed_netsloperound
```

2 Within the function body, create the table by calling the function `RTW.TflTable`.

```
hTable = RTW.TflTable;
```

3 Create the entry for the operator mapping with a call to the `RTW.TflCOperationEntryGenerator_Netslope` function, which provides access to the fixed-point parameters `NetSlopeAdjustmentFactor` and `NetFixedExponent`.

```
op_entry = RTW.TflCOperationEntryGenerator_NetSlope;
```

4 Set operator entry parameters with a call to the `setTflCOperationEntryParameters` function. The parameters specify the type of operation as division, the saturation mode as saturation off, rounding modes as round to ceiling, and the name of the replacement function as `s16_div_s16_s16`. `NetSlopeAdjustmentFactor` and `NetFixedExponent` specify the `F` and `E` parts of the relative scaling factor $F2^E$.

```
setTflCOperationEntryParameters(op_entry, ...
   'Key',                      'RTW_OP_DIV', ...
   'Priority',                 90, ...
   'SaturationMode',           'RTW_WRAP_ON_OVERFLOW', ...
   'RoundingModes',            {'RTW_ROUND_CEILING'}, ...
   'NetSlopeAdjustmentFactor',   1.0, ...
   'NetFixedExponent',           0.0, ...
```

```
    'ImplementationName',       's16_div_s16_s16', ...
    'ImplementationHeaderFile', 's16_div_s16_s16.h', ...
    'ImplementationSourceFile', 's16_div_s16_s16.c');
```

5 Create conceptual arguments `y1`, `u1`, and `u2`. There are multiple ways to set up the conceptual arguments. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. Specify each argument as fixed-point, 16 bits, and signed. Also, for each argument, specify that code replacement request processing does *not* check for an exact match to the call-site slope and bias values.

```
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);
```

6 Create the implementation arguments. There are multiple ways to set up the implementation arguments. This example uses calls to the `createAndSetCImplementationReturn` and `createAndAddImplementationArg` functions to create and add implementation arguments to the 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`).

```
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);
```

7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

8 Save the table definition file. Use the name of the table definition function to name the file.

More About

- "Code You Can Replace from MATLAB Code" on page 52-5
- "Define Code Replacement Mappings" on page 52-30
- "Fixed-Point Operator Code Replacement" on page 52-146
- "Data Type Conversions (Casts) and Operator Code Replacement" on page 52-169
- "Shift Left Operations and Code Replacement" on page 52-173
- "Remap Operator Output to Function Input" on page 52-143
- "Customize Match and Replacement Process" on page 52-112
- "Develop a Code Replacement Library" on page 52-15

Equal Slope and Zero Net Bias Code Replacement

You can define code replacement entries for addition or subtraction of fixed-point data types such that they match relative slope and bias values (equal slope and zero net bias) across operator inputs and output. These entries allow you to disregard slope and bias values. Map relative slope and bias values to a replacement function for addition or subtraction.

This example creates a code replacement entry for addition of fixed-point data types. Slopes must be equal and net bias must be zero across the operator inputs and output. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

1. Create a table definition file that contains a function definition. For example:

   ```
   function hTable = crl_table_fixed_slopeseq_netbiaszero
   ```

2. Within the function body, create the table by calling the function `RTW.TflTable`.

   ```
   hTable = RTW.TflTable;
   ```

3. Create the entry for the operator mapping with a call to the `RTW.TflCOperationEntryGenerator` function, which provides access to the fixed-point parameters `SlopesMustBeTheSame` and `MustHaveZeroNetBias`.

   ```
   op_entry = RTW.TflCOperationEntryGenerator;
   ```

4. Set operator entry parameters with a call to the `setTflCOperationEntryParameters` function. The parameters specify the type of operation as addition, the saturation mode as saturation off, rounding modes as unspecified, and the name of the replacement function as `u16_add_SameSlopeZeroBias`. `SlopesMustBeTheSame` and `MustHaveZeroNetBias` are set to `true`, indicating that slopes must be equal and net bias must be zero across the addition inputs and output.

   ```
   setTflCOperationEntryParameters(op_entry, ...
       'Key',                   'RTW_OP_ADD', ...
       'Priority',               90, ...
       'SaturationMode',         'RTW_WRAP_ON_OVERFLOW', ...
       'RoundingModes',          {'RTW_ROUND_UNSPECIFIED'}, ...
       'SlopesMustBeTheSame',    true, ...
       'MustHaveZeroNetBias',    true, ...
       'ImplementationName',      'u16_add_SameSlopeZeroBias', ...
       'ImplementationHeaderFile', 'u16_add_SameSlopeZeroBias.h', ...
       'ImplementationSourceFile', 'u16_add_SameSlopeZeroBias.c');
   ```

5 Create conceptual arguments `y1`, `u1`, and `u2`. There are multiple ways to set up the conceptual arguments. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. Each argument is specified as 16 bits and unsigned. Each argument specifies that code replacement request processing does *not* check for an exact match to the call-site slope and bias values.

```
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);
```

6 Create the implementation arguments. There are multiple ways to set up the implementation arguments. This example uses calls to the `createAndSetCImplementationReturn` and `createAndAddImplementationArg` functions to create and add implementation arguments to the 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`).

```
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);
```

7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

8 Save the table definition file. Use the name of the table definition function to name the file.

More About

- "Code You Can Replace from MATLAB Code" on page 52-5
- "Define Code Replacement Mappings" on page 52-30
- "Fixed-Point Operator Code Replacement" on page 52-146
- "Data Type Conversions (Casts) and Operator Code Replacement" on page 52-169
- "Shift Left Operations and Code Replacement" on page 52-173
- "Remap Operator Output to Function Input" on page 52-143
- "Customize Match and Replacement Process" on page 52-112
- "Develop a Code Replacement Library" on page 52-15

Data Type Conversions (Casts) and Operator Code Replacement

You can use code replacement entries to replace code that the code generator produces for data type conversion (cast) operations.

This example creates a code replacement entry that replaces `int32` to `int16` data type conversion (cast) operations. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

1. Create a table definition file that contains a function definition. For example:

   ```
   function hTable = crl_table_cast_int32_to_int16
   ```

2. Within the function body, create the table by calling the function `RTW.TflTable`.

   ```
   hTable = RTW.TflTable;
   ```

3. Create the entry for the operator mapping with a call to the `RTW.TflCOperationEntry` function.

   ```
   op_entry = RTW.TflCOperationEntry;
   ```

4. Set operator entry parameters with a call to the `setTflCOperationEntryParameters` function. The parameters specify the type of operation as cast, the saturation mode as saturate on integer overflow, rounding modes as toward negative infinity, and the name of the replacement function as `my_sat_cast`.

   ```
   setTflCOperationEntryParameters(op_entry, ...
       'Key',                    'RTW_OP_CAST', ...
       'Priority',               50, ...
       'ImplementationName',     'my_sat_cast', ...
       'SaturationMode',         'RTW_SATURATE_ON_OVERFLOW', ...
       'RoundingModes',          {'RTW_ROUND_FLOOR'}, ...
       'ImplementationHeaderFile', 'some_hdr.h', ...
       'ImplementationSourceFile', 'some_hdr.c');
   ```

5. Create the `int16` argument as conceptual argument `y1` and the implementation return value. There are multiple ways to set up the conceptual and implementation arguments. This example uses calls to the `getTflArgFromString` and `addConceptualArg` functions to create the conceptual argument and add it to the entry. Convenience method `setReturn` specifies the argument as the implementation return value.

   ```
   arg = getTflArgFromString(hTable, 'y1', 'int16');
   arg.IOType = 'RTW_IO_OUTPUT';
   addConceptualArg(op_entry, arg);
   ```

```
op_entry.Implementation.setReturn(arg);
```

6 Create the `int32` argument as conceptual and implementation argument `u1`. This example uses calls to the `getTflArgFromString` and `addConceptualArg` functions to create the conceptual argument and add it to the entry. Convenience method `addArgument` specifies the argument as implementation input argument.

```
arg = getTflArgFromString(hTable, 'u1', 'int32');
addConceptualArg(op_entry, arg);
op_entry.Implementation.addArgument(arg);
```

7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hLib, hEnt);
```

8 Save the table definition file. Use the name of the table definition function to name the file.

You can use code replacement entries to replace code that the code generator produces for data type conversion (cast) operations.

This example creates a code replacement entry to replace data type conversions (casts) of fixed-point data types by using a net slope. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

1 Create a table definition file that contains a function definition. For example:

```
function hTable = crl_table_cast_fixpt_net_slope
```

2 Within the function body, create the table by calling the function `RTW.TflTable`.

```
hTable = RTW.TflTable;
```

3 Create the entry for the operator mapping with a call to the `RTW.TflCOperationEntryGenerator_Netslope` function, which provides access to the fixed-point parameters `NetSlopeAdjustmentFactor` and `NetFixedExponent`

```
op_entry = RTW.TflCOperationEntryGenerator_NetSlope;
```

4 Set operator entry parameters with a call to the `setTflCOperationEntryParameters` function. The parameters specify the type of operation as cast, the saturation mode as saturate on integer overflow, rounding modes as toward negative infinity, and the name of the replacement function as `my_fxp_cast`. `NetSlopeAdjustmentFactor` and `NetFixedExponent` specify the `F` and `E` parts of the net slope $F2^E$.

```
InFL = 2;
InWL = 16;
InSgn = true;
OutFL = 4;
OutWL = 32;
OutSgn = true;
setTflCOperationEntryParameters(op_entry, ...
   'Key',                      'RTW_OP_CAST', ...
   'Priority',                 50, ...
   'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
   'RoundingModes',            {'RTW_ROUND_FLOOR'}, ...
   'NetSlopeAdjustmentFactor', 1.0, ...
   'NetFixedExponent',         (OutFL - InFL), ...
   'ImplementationName',       'my_fxp_cast', ...
   'ImplementationHeaderFile', 'some_hdr.h', ...
   'ImplementationSourceFile', 'some_hdr.c');
```

5 Create conceptual arguments `y1` and `u1`. There are multiple ways to set up the conceptual arguments. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. Each argument is specified as fixed-point and signed. Each argument specifies that code replacement request processing does *not* check for an exact match to the call-site slope and bias values.

```
createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
    'Name',         'y1', ...
    'IOType',       'RTW_IO_OUTPUT', ...
    'CheckSlope',   false, ...
    'CheckBias',    false, ...
    'DataTypeMode', 'Fixed-point: binary point scaling', ...
    'IsSigned',     OutSgn, ...
    'WordLength',   OutWL, ...
    'FractionLength',OutFL);

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
     'Name',         'u1', ...
     'IOType',       'RTW_IO_INPUT', ...
     'CheckSlope',   false, ...
     'CheckBias',    false, ...
     'DataTypeMode', 'Fixed-point: binary point scaling', ...
     'IsSigned',     InSgn, ...
     'WordLength',   InWL, ...
     'FractionLength',InFL);
```

6 Create the implementation arguments. There are multiple ways to set up the implementation arguments. This example uses calls to the `createAndSetCImplementationReturn` and `createAndAddImplementationArg` functions to create and add implementation arguments to the entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types).

```
createAndSetCImplementationReturn(op_entry, 'RTW.TflArgNumeric', ...
    'Name',           'y1', ...
    'IOType',         'RTW_IO_OUTPUT', ...
    'IsSigned',       OutSgn, ...
    'WordLength',     OutWL, ...
    'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
    'Name',           'u1', ...
    'IOType',         'RTW_IO_INPUT', ...
    'IsSigned',       InSgn, ...
    'WordLength',     InWL, ...
    'FractionLength', 0);
```

7 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

8 Save the table definition file. Use the name of the table definition function to name the file.

More About

- "Code You Can Replace from MATLAB Code" on page 52-5
- "Define Code Replacement Mappings" on page 52-30
- "Fixed-Point Operator Code Replacement" on page 52-146
- "Shift Left Operations and Code Replacement" on page 52-173
- "Remap Operator Output to Function Input" on page 52-143
- "Customize Match and Replacement Process" on page 52-112
- "Develop a Code Replacement Library" on page 52-15

Shift Left Operations and Code Replacement

You can use code replacement entries to replace code that the code generator produces for shift (<<) operations.

This example creates a code replacement entry to replace shift left operations for `int16` data. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

1. Create a table definition file that contains a function definition. For example:

   ```
   function hTable = crl_table_shift_left_int16
   ```

2. Within the function body, create the table by calling the function `RTW.TflTable`.

   ```
   hTable = RTW.TflTable;
   ```

3. Create the entry for the operator mapping with a call to the `RTW.TflCOperationEntry` function.

   ```
   op_entry = RTW.TflCOperationEntry;
   ```

4. Set operator entry parameters with a call to the `setTflCOperationEntryParameters` function. The parameters specify the type of operation as shift left and the name of the replacement function as `my_shift_left`.

   ```
   setTflCOperationEntryParameters(op_entry, ...
       'Key',                    'RTW_OP_SL', ...
       'Priority',               50, ...
       'ImplementationName',     'my_shift_left', ...
       'ImplementationHeaderFile', 'some_hdr.h', ...
       'ImplementationSourceFile', 'some_hdr.c');
   ```

5. Create the `int16` argument as conceptual argument `y1` and the implementation return value. There are multiple ways to set up the conceptual and implementation arguments. This example uses calls to the `getTflArgFromString` and `addConceptualArg` functions to create the conceptual argument and add it to the entry. Convenience method `setReturn` specifies the argument as the implementation return value.

   ```
   arg = getTflArgFromString(hTable, 'y1', 'int16');
   arg.IOType = 'RTW_IO_OUTPUT';
   addConceptualArg(op_entry, arg);
   op_entry.Implementation.setReturn(arg);
   ```

6. Create the `int16` argument as conceptual and implementation argument `u1`. This example uses calls to the `getTflArgFromString` and `addConceptualArg`

functions to create the conceptual argument and add it to the entry. Convenience method `addArgument` specifies the argument as an implementation input argument.

```
arg = getTflArgFromString(hTable, 'u1', 'int16');
addConceptualArg(op_entry, arg);
op_entry.Implementation.addArgument(arg);
```

7 Create the `int8` argument as conceptual and implementation argument `u2`. This example uses calls to the `getTflArgFromString` and `addConceptualArg` functions to create the conceptual argument and add it to the entry. This argument specifies the number of bits to shift the previous input argument. Because the argument type is not relevant, the example disables type checking by setting the `CheckType` property to `false`. Convenience method `addArgument` specifies the argument as implementation input argument.

```
arg = getTflArgFromString(hTable, 'u2', 'int8');
arg.CheckType = false;
addConceptualArg(op_entry, arg);
op_entry.Implementation.addArgument(arg);
```

- The function `getTflArgFromString` is called to create an `int8` input argument. This argument is added to the operator entry both as the third conceptual argument and the second implementation input argument.
- Add the entry to a code replacement table with a call to the `addEntry` function.

  ```
  addEntry(hTable, op_entry);
  ```
- Save the table definition file. Use the name of the table definition function to name the file.

You can use code replacement entries to replace code that the code generator produces for shift (<<) operations.

This example creates a code replacement entry to replace shift left operations for fixed-point data using a net slope. The example defines the function mapping programmatically. Alternatively, you can use the Code Replacement Tool to define the same mapping.

1 Create a table definition file that contains a function definition. For example:

```
function hTable = crl_table_shift_left_fixpt_net_slope
```

2 Within the function body, create the table by calling the function `RTW.TflTable`.

```
hTable = RTW.TflTable;
```

3 Create the entry for the operator mapping with a call to the `RTW.TflCOperationEntryGenerator_Netslope` function. This function provides access to the fixed-point parameters `NetSlopeAdjustmentFactor` and `NetFixedExponent`.

```
op_entry = RTW.TflCOperationEntryGenerator_NetSlope;
```

4 Set operator entry parameters with a call to the `setTflCOperationEntryParameters` function. The parameters specify the type of operation as shift left, the saturation mode as saturate on integer overflow, rounding modes as toward negative infinity, and the name of the replacement function as `my_fxp_shift_left`. `NetSlopeAdjustmentFactor` and `NetFixedExponent` specify the `F` and `E` parts of the net slope $F2^E$.

```
InFL = 2;
InWL = 16;
InSgn = true;
OutFL = 4;
OutWL = 32;
OutSgn = true;
setTflCOperationEntryParameters(op_entry, ...
   'Key',                      'RTW_OP_SL', ...
   'Priority',                 50, ...
   'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
   'RoundingModes',            {'RTW_ROUND_FLOOR'}, ...
   'NetSlopeAdjustmentFactor', 1.0, ...
   'NetFixedExponent',         (OutFL - InFL),...
   'ImplementationName',       'my_fxp_shift_left', ...
   'ImplementationHeaderFile', 'some_hdr.h', ...
   'ImplementationSourceFile', 'some_hdr.c');
```

5 Create conceptual arguments `y1` and `u1`. There are multiple ways to set up the conceptual arguments. This example uses calls to the `createAndAddConceptualArg` function to create and add an argument with one function call. Each argument is specified as fixed-point and signed. Each argument specifies that code replacement request processing does *not* check for an exact match to the call-site slope and bias values.

```
createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
    'Name',         'y1', ...
    'IOType',       'RTW_IO_OUTPUT', ...
    'CheckSlope',   false, ...
    'CheckBias',    false, ...
    'DataTypeMode', 'Fixed-point: binary point scaling', ...
    'IsSigned',     OutSgn, ...
    'WordLength',   OutWL, ...
    'FractionLength',OutFL);

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
    'Name',         'u1', ...
```

```
    'IOType',       'RTW_IO_INPUT', ...
    'CheckSlope',   false, ...
    'CheckBias',    false, ...
    'DataTypeMode', 'Fixed-point: binary point scaling', ...
    'IsSigned',     InSgn, ...
    'WordLength',   InWL, ...
    'FractionLength',InFL);
```

6 Create the implementation arguments. There are multiple ways to set up the implementation arguments. This example uses calls to the `createAndSetCImplementationReturn` and `createAndAddImplementationArg` functions to create and add implementation arguments to the entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types).

```
createAndSetCImplementationReturn(op_entry, 'RTW.TflArgNumeric', ...
    'Name',           'y1', ...
    'IOType',         'RTW_IO_OUTPUT', ...
    'IsSigned',       OutSgn, ...
    'WordLength',     OutWL, ...
    'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric', ...
     'Name',           'u1', ...
     'IOType',         'RTW_IO_INPUT', ...
     'IsSigned',       InSgn, ...
     'WordLength',     InWL, ...
     'FractionLength', 0);
```

7 Create the `int8` argument as conceptual and implementation argument `u2`. This example uses calls to the `getTflArgFromString` and `addConceptualArg` functions to create the conceptual argument and add it to the entry. This argument specifies the number of bits to shift the previous input argument. Because the argument type is not relevant, type checking is disabled by setting the `CheckType` property to `false`. Convenience method `addArgument` specifies the argument as implementation input argument.

```
arg = getTflArgFromString(hTable, 'u2', 'uint8');
arg.CheckType = false;
addConceptualArg(op_entry, arg);
op_entry.Implementation.addArgument(arg);
```

8 Add the entry to a code replacement table with a call to the `addEntry` function.

```
addEntry(hTable, op_entry);
```

9 Save the table definition file. Use the name of the table definition function to name the file.

More About

- "Code You Can Replace from MATLAB Code" on page 52-5
- "Define Code Replacement Mappings" on page 52-30
- "Fixed-Point Operator Code Replacement" on page 52-146
- "Data Type Conversions (Casts) and Operator Code Replacement" on page 52-169
- "Remap Operator Output to Function Input" on page 52-143
- "Customize Match and Replacement Process" on page 52-112
- "Develop a Code Replacement Library" on page 52-15

Performance

Optimizations for Generated Code in Simulink Coder

Increase Code Generation Speed

In this section...

The amount of time it takes to generate code for a model depends on the size and configuration settings of the model. For instance, if you are working with a large model, it can take awhile to generate code. To decrease the amount of time for code generation of a model, try one or more of the following methods:

- Build a model in increments
- Build large model reference hierarchies in parallel
- Minimize memory requirements during code generation
- Generate only code
- Disable the creation of a code generation report

Build a Model in Increments

You can use the `rtwbuild` (Simulink Coder) command to build a model and generate code. By default, when rebuilding a model, `rtwbuild` provides an incremental model build, which only rebuilds a model or submodels that have changed since the most recent model build. Incremental model build saves code generation time. Use the **Rebuild** parameter on the **Model Referencing** pane to change the method that Simulink uses to determine when to rebuild code for referenced models. For more information on the **Rebuild** parameter, see “Rebuild” (Simulink).

Build Large Model Reference Hierarchies in Parallel

In a parallel computing environment, whenever conditions allow, you can increase the speed of code generation and compilation by building models containing large model reference hierarchies in parallel. For example, if you have Parallel Computing Toolbox software, you can distribute code generation and compilation for each referenced

model across the cores of a multicore host computer. If you have MATLAB Distributed Computing Server software, you can distribute code generation and compilation for each referenced model across remote workers in your MATLAB Distributed Computing Server configuration.

The performance gain realized by using parallel builds for referenced models depends on several factors, including:

- How many models can be built in parallel for a given model referencing hierarchy
- The size of the referenced models
- Parallel computing resources such as the number of local and remote workers available
- The hardware attributes of the local and remote machines (amount of RAM, number of cores, and so on)

For more information, see "Reduce Build Time for Referenced Models" (Simulink Coder).

Minimize Memory Requirements During Code Generation

Models that have large amounts of parameter and constant data (such as lookup tables) can tax memory resources and slow code generation. The code generator copies this data to the *`model`*`.rtw` file. The *`model`*`.rtw` file is a partial representation of the model that the Target Language Compiler parses to transform block computations, parameters, signals, and constant data into a high-level language (for example, C). The Target Language Compiler (TLC) is an integral part of the code generator. The code generator copies parameters and data into *`model`*`.rtw`, whether they originate in the model or come from variables or objects in a workspace.

You can improve code generation speed by specifying the maximum number of elements that data vectors can have for the code generator to copy this data to *`model`*`.rtw`. When a data vector exceeds the specified size, the code generator places a reference key in *`model`*`.rtw`. The TLC uses this key to access the data from Simulink and format it into the generated code. Reference keys result in maintaining only one copy of large data vectors in memory.

The default value above which the code generator uses reference keys in place of actual data values is 10 elements. You can verify this value. In the Command Window, type the following command:

```
get_param(0, 'RTWDataReferencesMinSize')
```

To set the threshold to a different value, in the Command Window, type the following `set_param` function:

```
set_param(0, 'RTWDataReferencesMinSize', <size>)
```

Provide an integer value for `size` that specifies the number of data elements above which the code generator uses reference keys in place of actual data values.

Generate Only Code

You can increase code generation speed by specifying that the build process generate code and a makefile, but not invoke the make command. When the code generator invokes the make command, the build process takes longer because the code generator generates code, compiles code, and creates an executable file.

On the **Code Generation** pane in the Model Configuration Parameters dialog box, you can specify that the build process generate only code by selecting the **Generate code only** parameter. You can specify that the code generation process build a makefile by selecting the **Generate makefile** parameter on the **All Parameters** tab.

No Creation of a Code Generation Report

You can speed up code generation by not generating a code generation report as a part of the build process. To disable the creation of a code generation report, on the **Code Generation** > **Report** pane, clear the **Create code generation report** parameter. After the build process, you can generate a code generation report by doing this procedure, “Generate Code Generation Report After Build Process” (Simulink Coder).

Related Examples

- “Enable parallel model reference builds” (Simulink)
- “MATLAB worker initialization for builds” (Simulink)
- “Reduce Build Time for Referenced Models” on page 40-50

Control Compiler Optimizations

To control compiler optimizations for a makefile build at the GUI level, use the **Compiler optimization level** parameter. The **Compiler optimization level** parameter provides

- Target-independent values `Optimizations on (faster runs)` and `Optimizations off (faster builds)`, which allow you to easily toggle compiler optimizations on and off during code development
- The value `Custom` for entering custom compiler optimization flags at the Simulink GUI level, rather than editing compiler flags into template makefiles (TMFs) or supplying compiler flags to build process make commands

The default setting is `Optimizations off (faster builds)`. Selecting the value `Custom` enables the **Custom compiler optimization flags** field, in which you can enter custom compiler optimization flags (for example, `-O2`).

Note: If you specify compiler options for your makefile build using `OPT_OPTS`, `MEX_OPTS` (except `MEX_OPTS="-v"`), or `MEX_OPT_FILE`, the value of **Compiler optimization level** is ignored and a warning is issued about the ignored parameter.

For more information about the **Compiler optimization level** parameter and its values, see "Compiler optimization level" (Simulink Coder) and "Custom compiler optimization flags" (Simulink Coder).

Related Examples

- "Template Makefiles and Make Options" on page 40-24
- "Select a System Target File" on page 30-2
- "Support Compiler Optimization Level Control" on page 71-95

Optimization Tools and Techniques

Use the Model Advisor to Optimize a Model for Code Generation

You can use the Model Advisor to analyze a model for code generation and identify aspects of your model that impede production deployment or limit code efficiency. You can select from a set of checks to run on a model's current configuration. The Model Advisor analyzes the model and generates check results providing suggestions for improvements in each area. Most Model Advisor diagnostics do not require the model to be in a compiled state; those that do are noted.

Before running the Model Advisor, select the target you plan to use for code generation. The Model Advisor works most effectively with ERT and ERT-based system target files.

Use the following examples to investigate optimizing models for code generation using the Model Advisor:

- `rtwdemo_advisor1`
- `rtwdemo_advisor2`
- `rtwdemo_advisor3`

Note: Example models `rtwdemo_advisor2` and `rtwdemo_advisor3` require Stateflow and Fixed-Point Designer software.

For more information on using the Model Advisor, see “Run Model Checks” (Simulink). For more information about the checks, see “Simulink Coder Checks” (Simulink Coder).

Design Tips for Optimizing Generated Code for Stateflow Objects

Do Not Access Machine-Parented Data In a Graphical Function

This restriction prevents long parameter lists from appearing in the code generated for a graphical function. You can access local data that resides in the same chart as the graphical function.

For more information, see “Reuse Logic Patterns Using Graphical Functions” (Stateflow).

Be Explicit About the Inline Option of a Graphical Function

When you use a graphical function in a Stateflow chart, select `Inline` or `Function` for the property **Function Inline Option**. Otherwise, the code generated for a graphical function may not appear as you want.

For more information, see "Specify Graphical Function Properties" (Stateflow).

Avoid Using Multiple Edge-Triggered Events in Stateflow Charts

When you use a bus object, you reduce the number of parameters in the parameter list of a generated function. This guideline also applies to output signals of a chart.

For more information, see "Define Stateflow Structures" (Stateflow).

Combine Input Signals of a Chart Into a Single Bus Object

When you use a bus object, you reduce the number of parameters in the parameter list of a generated function. This guideline also applies to output signals of a chart.

For more information, see "Define Stateflow Structures" (Stateflow).

Use Charts with Discrete Sample Times

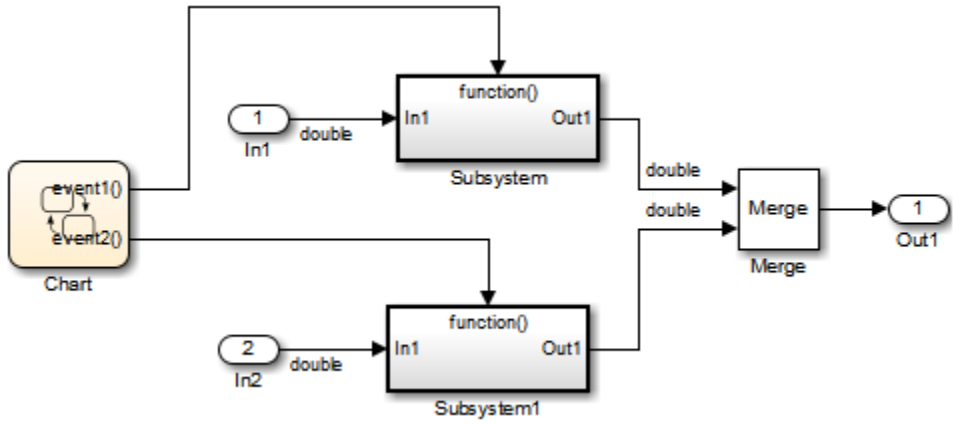
The code generated for discrete charts that are not inside a triggered or enabled subsystem uses integer counters to track time instead of Simulink provided time. This allows for more efficient code generation in terms of overhead and memory, as well as enabling this code for use in Software-in-the-Loop(SIL) and Processor-in-the-Loop(PIL) simulation modes.

Additional Optimization Techniques

You can apply the following techniques to optimize a model for code generation:

- Use the Upgrade Advisor to upgrade older models (saved by prior versions or the current version) to use current features. For details, see "Model Upgrades" (Simulink).
- Before building, set optimization flags for the compiler (for example, `-O2` for `gcc`, `-Ot` for the Microsoft Visual C++ compiler).
- Directly inline C/C++ S-functions into the generated code by writing a TLC file for the S-function. For more information, see "Accelerate Simulation, Reuse Code, or Protect Intellectual Property by Using S-Function Target" (Simulink Coder) and see "Inline C MEX S-Functions" (Simulink Coder).

- Use a Simulink data type other than `double` when possible. The available data types are `Boolean`, signed and unsigned 8-, 16-, and 32-bit integers, and 32- and 64-bit floats (a `double` is a 64-bit float). For more information, see "About Data Types in Simulink" (Simulink). For a block-by-block summary, click showblockdatatypetable or type the command in the Command Window.
- For tunable block parameters that you configure to store in memory in the generated code, you can match parameter data types with signal data types to eliminate unnecessary typecasts and C shifts. Where possible, store parameter values in small integer data types. See "Parameter Data Types in the Generated Code" on page 19-79.
- Remove repeated values in lookup table data.
- Use the Merge block to merge the output of signals wherever possible. This block is particularly helpful when you need to control the execution of function-call subsystems with a Stateflow chart. The following model shows an example of how to use the Merge block.



When more than one signal connected to a Merge block has a non-`Auto` storage class, all non-`Auto` signals connected to that block must *be identically labeled* and *have the same storage class*. When Merge blocks connect directly to one another, these rules apply to the signals connected to any of the Merge blocks in the group.

Related Examples

- "Increase Code Generation Speed" on page 53-3
- "Execution Profiling for Generated Code" on page 53-12

- “Optimization Pane: General” (Simulink)
- “Optimization Pane: Signals and Parameters” (Simulink)
- “Model Configuration Parameters: Advanced Parameters” (Simulink Coder)

Control Memory Allocation for Time Counters

The **Application lifespan (days)** parameter lets you control the allocation of memory for absolute and elapsed time counters. Such counters exist in the code for blocks that use absolute or elapsed time. For a list of such blocks, see "Absolute Time Limitations" (Simulink Coder).

The size of the time counters in generated code is 8, 16, 32, or 64 bits. The size is set automatically to the minimum that can accommodate the duration value specified by **Application lifespan (days)** given the step size specified in the Configuration Parameters **Solver** pane. To minimize the amount of RAM used by time counters, specify the smallest lifespan possible and the largest step size possible.

An application runs to its specified lifespan. It may be able to run longer. For example, running a model with a step size of one millisecond (0.001 seconds) for one day requires a 32-bit timer, which could continue running without overflow for 49 days more.

To maximize application lifespan, specify **Application lifespan (days)** as `inf`. This value allocates 64 bits (two `uint32` words) for each timer. Using 64 bits to store timing data would allow a model with a step size of 0.001 microsecond (10E-09 seconds) to run for more than 500 years, which would rarely be required. 64-bit counters do not violate the usual code generator length limitation of 32 bits because the value of a time counter does not provide the value of a signal, state, or parameter.

See Also

"Application lifespan (days)" (Simulink)

Related Examples

- "Absolute and Elapsed Time Computation" (Simulink Coder)
- "Timers in Asynchronous Tasks" (Simulink Coder)

Execution Profiling for Generated Code

Use code execution profiling to:

- Determine whether the generated code meets execution time requirements for real-time deployment on your target hardware.
- Identify code sections that require performance improvements.

The following tasks represent a general workflow that uses code execution profiling:

1. With the Simulink model, design and optimize your algorithm.
2. Configure the model for code execution profiling, and generate code.
3. Execute generated code on target. For example, you can:
   - Run a software-in-the-loop (SIL) simulation on your development computer.
   - Run a processor-in-the-loop (PIL) simulation using a target support package or custom PIL target.
   - Perform real-time execution with Simulink Real-Time or a target support package.
4. Analyze performance through code execution profiling plots and reports. For example, check that the algorithm code satisfies execution time requirements for real-time deployment:
   - If the algorithm code easily meets the requirements, consider enhancing your algorithm to exploit available processing power.
   - If the code cannot be executed in real time, look for ways to reduce execution time.

     Identify the tasks that require the most time. For these tasks, investigate whether trade-offs between functionality and speed are possible.

     If your target is a multicore processor, distribute the execution of algorithm code across available cores.
5. If required, refine the model and return to step 2.

To find information about code execution profiling with Simulink products, use the following table.

Target	Execution Feature	Type of Profiling	Relevant Products	See
Development computer	Model configured for concurrent execution	Execution time	Simulink Coder	• "Optimize and Deploy on a Multicore Target" (Simulink) • "Concurrent Execution Models" (Simulink)
Development computer	Software-in-the-loop (SIL)	Execution time	Embedded Coder	• "Code Execution Profiling with SIL and PIL" on page 58-2 • "View and Compare Code Execution Times" on page 58-7 • "Analyze Code Execution Data" on page 58-18
Embedded hardware or instruction set simulator	Processor-in-the-loop (PIL)	Execution time	Embedded Coder	• "Code Execution Profiling with SIL and PIL" on page 58-2 • "View and Compare Code Execution Times" on page 58-7 • "Analyze Code Execution Data" on page 58-18
Target support packages	Real-time execution, PIL	Execution time	Embedded Coder	• "Code Execution Profiling for IDE and Toolchain Targets" on page 73-13 • "Perform Execution-Time Profiling for IDE and Toolchain Targets" on page 73-16
Target support packages	Real-time execution	Stack	Embedded Coder	• "Code Execution Profiling for IDE and Toolchain Targets" on page 73-13 • "Perform Stack Profiling with IDE and Toolchain Targets" on page 73-22

Target	Execution Feature	Type of Profiling	Relevant Products	See
Simulink Real-Time	Real-time execution	Execution time	Simulink Coder, Simulink Real-Time	• "Execution Profiling for Real-Time Applications" (Simulink Real-Time) • "Configure Real-Time Application for Profiling" (Simulink Real-Time) • "Generate Real-Time Application Execution Profile" (Simulink Real-Time)
Simulink Real-Time	Real-time execution, model configured for concurrent execution	Execution time	Simulink Coder, Simulink Real-Time	• "Execution Profiling for Real-Time Applications" (Simulink Real-Time) • "Concurrent Execution on Simulink® Real-Time™" (Simulink Real-Time)

Related Examples

- "Optimization Tools and Techniques" on page 53-7
- "SIL and PIL Simulations" on page 64-2

Optimize Generated Code by Combining Multiple `for` Constructs

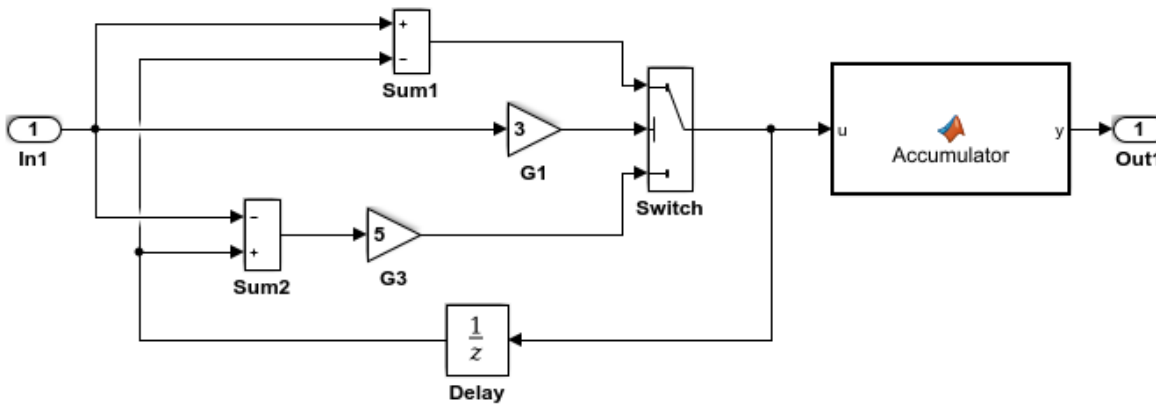
This example shows how the code generator combines `for` loops. The generated code uses `for` constructs to represent a variety of modeling patterns, such as a matrix signal or Iterator blocks. Using data dependency analysis, the code generator combines `for` constructs to reduce static code size and runtime branching.

The benefits of optimizing `for` loops are:

- Reducing ROM and RAM consumption.
- Increasing execution speed.

`for` Loop Modeling Patterns

In the model, rtwdemo_forloop, the Switch block and MATLAB Function block represent `for` constructs. In the `In1` Block Parameters dialog box, the **Port dimensions** parameter is set to `10` .



This model shows how Simulink Coder optimizes for-loops by combining multiple blocks into the same for-loop, improving code efficiency and readability. The for-loop fusion optimization complements expression folding. To see the vector widths in this model, select Edit > Update Diagram.

Generate Code Using Simulink Coder (double-click)

Generate Code Using Embedded Coder (double-click)

Generate Code

In the model, there are no data dependencies across the `for` loop iterations. Therefore, the code generator combines all `for` loops into one loop. Build the model and view the generated code.

```
### Starting build procedure for model: rtwdemo_forloop
### Successful completion of build procedure for model: rtwdemo_forloop
```

The generated file, `rtwdemo_forloop.c`, contains the code for the single `for` loop.

```
/* Model step function */
void rtwdemo_forloop_step(void)
{
  int32_T k;

  /* MATLAB Function: '<Root>/Accum' */
  /* MATLAB Function 'Accum': '<S1>:1' */
  /* '<S1>:1:3' */
  /* '<S1>:1:4' */
  rtwdemo_forloop_Y.Out1 = 0.0;

  /* '<S1>:1:5' */
  for (k = 0; k < 10; k++) {
    /* Switch: '<Root>/Switch' incorporates:
     *  Gain: '<Root>/G1'
     *  Gain: '<Root>/G3'
     *  Inport: '<Root>/In1'
     *  Sum: '<Root>/Sum1'
     *  Sum: '<Root>/Sum2'
     *  UnitDelay: '<Root>/Delay'
     */
    if (3.0 * rtwdemo_forloop_U.In1[k] >= 0.0) {
      rtwdemo_forloop_DW.Delay_DSTATE[k] = rtwdemo_forloop_U.In1[k] -
        rtwdemo_forloop_DW.Delay_DSTATE[k];
    } else {
      rtwdemo_forloop_DW.Delay_DSTATE[k] = (rtwdemo_forloop_DW.Delay_DSTATE[k] -
        rtwdemo_forloop_U.In1[k]) * 5.0;
    }

    /* End of Switch: '<Root>/Switch' */

    /* MATLAB Function: '<Root>/Accum' */
    /* '<S1>:1:5' */
```

```
    /* '<S1>:1:6' */
    rtwdemo_forloop_Y.Out1 += (1.0 + (real_T)k) +
      rtwdemo_forloop_DW.Delay_DSTATE[k];
  }
}
```

Close the model.

Related Examples

- "Optimization Tools and Techniques" on page 53-7
- "Configure Loop Unrolling Threshold" on page 53-49
- "For Loop" on page 13-40

Subnormal Number Performance

Subnormal numbers, formerly known as denormal numbers in floating-point literature, fill the underflow gap around zero in floating-point arithmetic. Subnormal values are a special category of floating-point values that are too close to `0.0` to be represented as a normalized value. The leading significand (mantissa) of a subnormal number is zero. When adding and subtracting floating-point numbers, subnormal numbers prevent underflow.

Using subnormal numbers provides precision beyond the normal representation by using leading zeros in the significand to represent smaller values after the representation reaches the minimum exponent. As the value approaches `0.0`, you trade off precision for extended range. Subnormal numbers are useful if your application requires extra range.

However, in a real-time system, using subnormal numbers can dramatically increase execution latency, resulting in excessive design margins and real-time overruns. If the simulation or generated code performs calculations that produce or consume subnormal numbers, the execution of these calculations can be up to 50 times slower than similar calculations on normal numbers. The actual simulation or code execution time for subnormal number calculations depends on your computer operating environment. Typically, for desktop processors, the execution time for subnormal number calculations is five times slower than similar calculations on normal numbers.

To minimize the possibility of execution slowdowns or overruns due to subnormal number calculation latency, do one of the following:

- In your model, manually flush to zero any incoming or computed subnormal values at inputs and key operations, such as washouts and filters. For an example, see “Flush Subnormal Numbers to Zero” on page 53-20.

  To detect a subnormal value for a single precision, 32-bit floating-point number:

  1. Find the smallest normalized number on a MATLAB host. In the Command Window, type:

     ```
     >> SmallestNormalSingle = realmin('single')
     ```

     In the C language, `FLT_MIN` , defined in `float.h`, is equivalent to `realmin('single')`.

  2. Look for values in range:

     ```
     0 < fabsf(x) < SmallestNormalSingle
     ```

To detect a subnormal value for a double precision, 64-bit floating-point number:

1. Find the smallest normalized number on a MATLAB host. In the Command Window, type:

   ```
   >> SmallestNormalDouble = realmin('double')
   ```
   In the C language, `DBL_MIN` , defined in `float.h`, is equivalent to `realmin('double')`.

2. To detect a subnormal value, look for values in this range:

   ```
   0 < fabs(x) < SmallestNormalDouble
   ```

- On your processor, set flush-to-zero mode or, with your compiler, specify an option to disable subnormal numbers. Flush-to-zero modes treat a subnormal number as 0 when it is an input to a floating-point operation. Underflow exceptions do not occur in flush-to-zero mode.

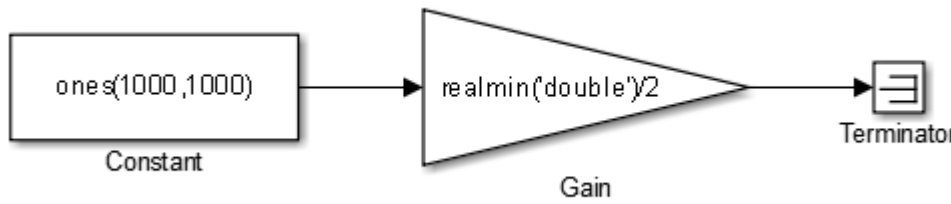
  For example, in Intel® processors, the flush-to-zero (FTZ) and denormals-are-zero (DAZ) flags in the MXCSR register control floating-point calculations. For the gcc compiler on Linux, `-ffast-math` sets abrupt underflow (FTZ), flush-to-zero, while `-O3 -ffast-math` reverts to gradual underflow, using subnormal numbers.

For more information, see the IEEE Standard 754, *IEEE Standard for Floating-Point Arithmetic*.

Simulation Time With and Without Subnormal Numbers

This model shows how using subnormal numbers increases simulation time by ~5 times.

1. Open the model ex_subnormal. The Gain is set to subnormal value `realmin('double')/2`.



2 To run a simulation, in the Command Window, type `for k=1:5, tic; sim('ex_subnormal'); toc,end`. Observe the elapsed times for simulation using subnormals, similar to the following:

```
>> for k=1:5, tic; sim('ex_subnormal'); toc,end
Elapsed time is 9.909326 seconds.
Elapsed time is 9.617966 seconds.
Elapsed time is 9.797183 seconds.
Elapsed time is 9.702397 seconds.
Elapsed time is 9.893946 seconds.
```

3 Set the Gain to a number, 2, that is not a subnormal value:

```
>> set_param('ex_subnormal/Gain', 'Gain', '2');
```

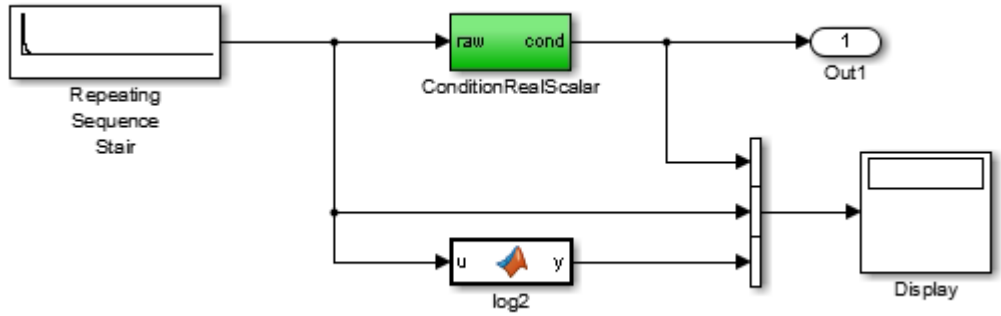
4 To run a simulation, in the Command Window, type `for k=1:5, tic; sim('ex_subnormal'); toc,end`. Observe elapsed times for simulations that do not use subnormal values, similar to the following:

```
>> for k=1:5, tic; sim('ex_subnormal'); toc,end
Elapsed time is 2.045123 seconds.
Elapsed time is 1.796598 seconds.
Elapsed time is 1.758458 seconds.
Elapsed time is 1.721721 seconds.
Elapsed time is 1.780569 seconds.
```

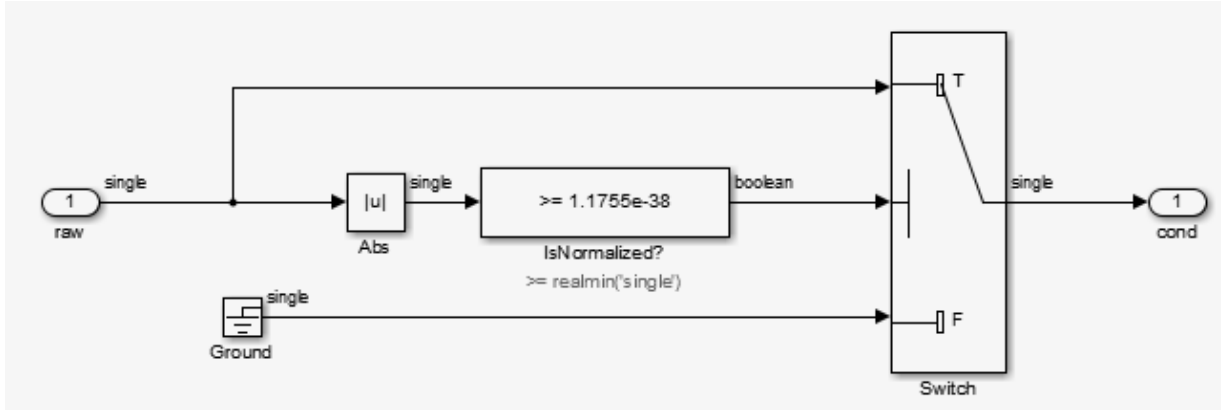
Flush Subnormal Numbers to Zero

This example shows how to flush single precision subnormal numbers to zero.

1 Open the model ex_flush_to_zero:



- `Repeating Sequence Stair` generates a sequence of numbers from two raised to the power of 0 through two raised to the power of -165. The sequence approaches zero.
- `ConditionRealScalar` flushes subnormal single precision values that are less than `realmin('single')` to zero.



- MATLAB function block `log2` generates the base 2 logarithm of the `Repeating Sequence Stair` output. Specifically, `log2` generates the numbers 0 through -165.

2 On the **Simulation > Stepping Options** pane:

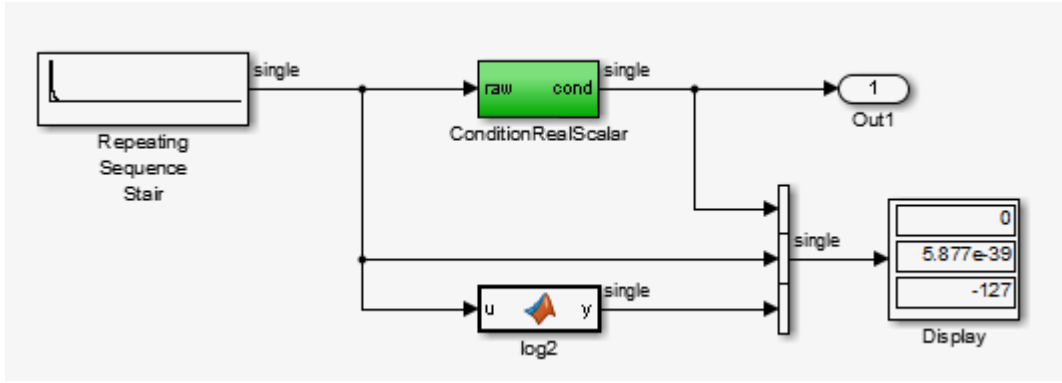
- Select **Enable stepping back**.

- Select **Pause simulation when time reaches** and enter `121`.

3 In the model window, run the simulation. The simulation pauses at `T=121`. The displayed values:

- `ConditionRealScalar` output approaches zero.
- `Repeating Sequence Stair output` approaches zero.

4 Step the simulation forward to `T=127`. `ConditionRealScalar` flushes the subnormal value output from `Repeating Sequence Stair` to zero.



5 Continue stepping the simulation forward. `ConditionRealScalar` flushes the subnormal single precision values output from `Repeating Sequence Stair` to zero. When `T=150`, the output of `Repeating Squence Stair` is itself zero.

Related Examples

- "Data Types Supported by Simulink" (Simulink)
- "Numerical Consistency of Model and Generated Code Simulation Results" (Simulink Coder)
- "Specify Single-Precision Data Type for Embedded Application" on page 19-43

Remove Code From Floating-Point to Integer Conversions That Wraps Out-of-Range Values

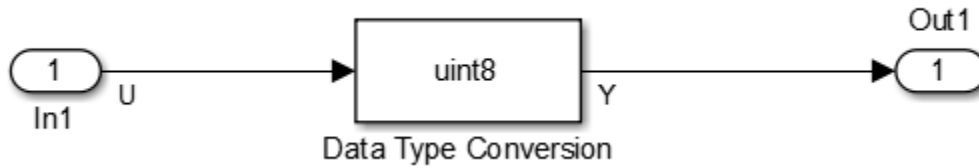
In this section...

This example shows how to remove code for out-of-range floating-point to integer conversions. Without this code, there might be a mismatch between simulation and code generation results. Standard C does not define the behavior of out-of-range floating-point to integer conversions, while these conversions are well-defined during simulation. In Standard C and during simulation, floating-point to integer conversions are well-defined for input values in the range of the output type.

If the input values in your application are in the range of the output type, remove code for out-of-range floating-point to integer conversions. Removing this code reduces the size and increases the speed of the generated code.

Example Model

In this model, a Data Type Conversion block converts an input signal from a `double` to a `uint8`. A `uint8` can support values from 0 to 255. If the input signal has a value outside of this range, an out-of-range conversion occurs. In this example, the model is named `conversion_ex`.



1 Use Inport, Outport, and Data Type Conversion blocks to create the example model.

2. Open the Inport Block Parameters dialog box and select the **Signal Attributes** tab. For the **Data Type** parameter, select `double`.
3. Open the Data Type Conversion dialog box. For the **Output data type** parameter, select `uint8`.
4. For the signal feeding into the Data Type Conversion block, open the Signal Properties dialog box. Enter the name `U`. On the **Code Generation** tab, for the **Storage Class** parameter, select `ImportedExtern`.
5. For the signal leaving the Data Type Conversion block, open the Signal Properties dialog box. Enter the name `Y`. On the **Code Generation** tab, for the **Storage Class** parameter, select `ImportedExtern`.

Generate Code Without Optimization

1. Open the Model Configuration Parameters dialog box. On the **Solver** pane, for the **Type** parameter, select `Fixed-step`.
2. On the **Code Generation > Report** pane, select **Create code generation report**.
3. On the **Code Generation** pane, select **Generate code only**, and then, in the model window, press **Ctrl+B**. When code generation is complete, an HTML code generation report opens.
4. In the Code Generation report, select the `conversion_ex.c` file and view the model step function. The code generator applies the `fmod` function to handle out-of range-results.

```
/* Model step function */
void conversion_ex_step(void)
{
  real_T tmp;

  /* DataTypeConversion: '<Root>/Data Type Conversion' incorporates:
   *  Inport: '<>/In1'
   */
  tmp = floor(U);
  if (rtIsNaN(tmp) || rtIsInf(tmp)) {
    tmp = 0.0;
  } else {
    tmp = fmod(tmp, 256.0);
  }

  Y = (uint8_T)(tmp < 0.0 ? (int32_T)(uint8_T)-(int8_T)(uint8_T)-tmp : (int32_T)
                (uint8_T)tmp);
```

Generate Code with Optimization

1. Open the Configuration Parameters dialog box. On the **Optimization** pane, select **Remove code from floating-point to integer conversions that wraps out-of-range values**. Generate code.
2. In the code generation report, select the `conversion_ex.c` file and view the model step function. The generated code does not contain code that protects against out-of-range values.

```
/* Model step function */
void conversion_ex_step(void)
{
  /* DataTypeConversion: '<Root>/Data Type Conversion' incorporates:
   *  Inport: '<Root>/In1'
   */
  Y = (uint8_T)U;
```

The generated code is more efficient without this protective code, but it is possible that the execution of generated code does not produce the same results as simulation for values not in the range of 0 to 255.

See Also

"Remove code from floating-point to integer conversions that wraps out-of-range values" (Simulink)

Related Examples

- "hisl_0053: Configuration Parameters > Optimization > Remove code from floating-point to integer conversions that wraps out-of-range values" (Simulink)
- "Optimization Tools and Techniques" on page 53-7
- "Remove Code That Maps NaN to Integer Zero" on page 53-26

Remove Code That Maps NaN to Integer Zero

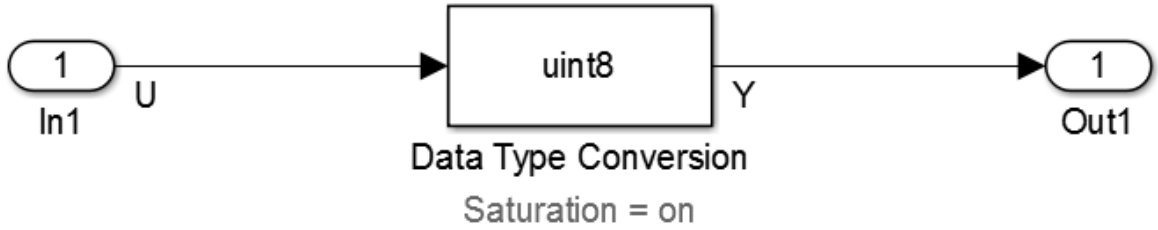
In this section...

This example shows how to remove code that maps `NaN` to integer zero. For floating-point to integer conversions involving saturation, Simulink converts `NaN` to integer zero during simulation. If your model contains an input value of `NaN`, you can specify that the code generator produce code that maps `NaN` to zero. Without this code, there is a mismatch between simulation and code generation results because in Standard C, every condition involving `NaN` evaluates to false.

If there are no input values of `NaN` in your application, you can remove code that maps `NaN` to integer zero. Removing this code reduces the size and increases the speed of the generated code.

Example Model

In this model, a Data Type Conversion block converts an input signal from a `double` to a `uint8`. In this example, the model is named `conversion_ex`.



1. Use Inport, Outport, and Data Type Conversion blocks to create the example model.
2. Open the Inport Block Parameters dialog box and click the **Signal Attributes** tab. For the **Data Type** parameter, select `double`.
3. Open the Data Type Conversion dialog box. For the **Output data type** parameter, select `uint8`.

4. Select **Saturate on integer overflow**. Selecting this parameter specifies that an out-of-range signal value equals either the minimum or maximum value that the data type can represent.
5. For the signal feeding into the Data Type Conversion block, open the Signal Properties dialog box. Enter a name of `U`. On the **Code Generation** tab, for the **Storage Class** parameter, select `ImportedExtern`.
6. For the signal leaving the Data Type Conversion block, open the Signal Properties dialog box. Enter a name of `Y`. On the **Code Generation** tab, for the **Storage Class** parameter, select `ImportedExtern`.

Generate Code

1. Open the Model Configuration Parameters dialog box. On the **Solver** pane, for the **Type** parameter, select `Fixed-step` .
2. Open the Model Configuration Parameters dialog box. On the **All Parameters** tab, clear **Remove code from floating-point to integer conversions with saturation that maps NaN to zero**.
3. On the **Code Generation** > **Report** pane, select **Create code generation report**.
4. On the **Code Generation** pane, select **Generate code only** and then, in the model window, press **Ctrl+B**. When code generation is complete, an HTML code generation report opens.
5. In the Code Generation report, select the `nan_int_ex.c` file and view the model step function. For an input value of `NaN`, there is agreement between the generated code and simulation because `NaN` maps to integer zero.

```
/* Model step function */
void nan_int_ex_step(void)
{
  /* DataTypeConversion: '<Root>/Data Type Conversion' incorporates:
   *  Inport: '<Root>/In1'
   */
  if (U < 256.0) {
    if (U >= 0.0) {
      Y = (uint8_T)U;
    } else {
      Y = 0U;
    }
  } else if (U >= 256.0) {
```

```
    Y = MAX_uint8_T;
  } else {
    Y = 0U;
  }
```

Generate Code with Optimization

1. Open the Configuration Parameters dialog box. On the **All Parameters** tab, select **Remove code from floating-point to integer conversions that wraps out-of-range values**. Generate code.
2. In the Code Generation report, select the `nan_int_ex.c` section and view the model step function. The generated code maps `NaN` to 255 and not integer zero. The generated code is more efficient without the extra code that maps `NaN` to integer zero, but the execution of the generated code does not produce the same results as simulation for `NaN` values.

```
/* Model step function */
void nan_int_ex_step(void)
{
  /* DataTypeConversion: '<Root>/Data Type Conversion' incorporates:
   *  Inport: '<Root>/In1'
   */
  if (U < 256.0) {
    if (U >= 0.0) {
      Y = (uint8_T)U;
    } else {
      Y = 0U;
    }
  } else {
    Y = MAX_uint8_T;
  }

  /* End of DataTypeConversion: '<Root>/Data Type Conversion' */
```

See Also

"Remove code from floating-point to integer conversions with saturation that maps NaN to zero" (Simulink)

Related Examples

- "Optimization Tools and Techniques" on page 53-7

- “Remove Code From Floating-Point to Integer Conversions That Wraps Out-of-Range Values” on page 53-23

Disable Nonfinite Checks or Inlining for Math Functions

When the code generator produces code for math functions:

- If the model option **Support non-finite numbers** is selected, nonfinite number checking is generated uniformly for math functions, without the ability to specify that nonfinite number checking should be generated for some functions, but not for others.
- By default, inlining is applied uniformly for math functions, without the ability to specify that inlining should be generated for some functions, while invocations should be generated for others.

You can use code replacement library (CRL) customization entries to:

- Selectively disable nonfinite checks for math functions. This can improve the execution speed of the generated code.
- Selectively disable inlining of math functions. This can increase code readability and decrease code size.

The functions for which these customizations are supported include the following:

- Floating-point only: `atan2`, `copysign`, `fix`, `hypot`, `log`, `log10`, `round`, `sincos`, and `sqrt`
- Floating-point and integer: `abs`, `max`, `min`, `mod`, `rem`, `saturate`, and `sign`

The general workflow for disabling nonfinite number checking and/or inlining is as follows:

1. If you can disable nonfinite number checking for a particular math function, or if you want to disable inlining for a particular math function and instead generate a function invocation, you can copy the following MATLAB function code into a MATLAB file with an `.m` file name extension, for example, `crl_table_customization.m`.

```
function hTable = crl_table_customization

% Create an instance of the Code Replacement Library table for controlling
% function intrinsic inlining and nonfinite support

hTable = RTW.TflTable;

% Inline - true (if function needs to be inline)
%          false (if function should not be inlined)
% SNF (support nonfinite) - ENABLE (if non-finite checking should be performed)
%                           DISABLE (if non-finite checking should NOT be performed)
%                           UNSPECIFIED (Default behavior)
```

```
% registerCustomizationEntry(hTable, ...
%       Priority, numInputs, key, inType, outType, Inline, SNF);

registerCustomizationEntry(hTable, ...
        100, 2, 'atan2', 'double', 'double', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'atan2', 'single', 'single', false, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
        100, 1, 'sincos', 'double', 'double', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'sincos', 'single', 'single', false, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
        100, 1, 'abs', 'double', 'double', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'abs', 'single', 'single', true, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
        100, 1, 'abs', 'int32',  'int32',  true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'abs', 'int16',  'int16',  true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'abs', 'int8',   'int8',   true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'abs', 'integer','integer',true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'abs', 'uint32', 'uint32', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'abs', 'uint16', 'uint16', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'abs', 'uint8',  'uint8',  true, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
        100, 2, 'hypot', 'double', 'double', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'hypot', 'single', 'single', false, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
        100, 1, 'log', 'double', 'double', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'log', 'single', 'double', false, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
        100, 1, 'log10', 'double', 'double', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'log10', 'single', 'double', false, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
        100, 2, 'min', 'double', 'double', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'min', 'single', 'single', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'min', 'int32',  'int32',  true, 'UNSPECIFIED');
```

```
registerCustomizationEntry(hTable, ...
        100, 2, 'min', 'int16',  'int16',  true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'min', 'int8',   'int8',   true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'min', 'uint32', 'uint32', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'min', 'uint16', 'uint16', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'min', 'uint8',  'uint8',  true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'min', 'integer','integer',true, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
        100, 2, 'max', 'double', 'double', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'max', 'single', 'single', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'max', 'int32',  'int32',  true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'max', 'int16',  'int16',  true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'max', 'int8',   'int8',   true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'max', 'uint32', 'uint32', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'max', 'uint16', 'uint16', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'max', 'uint8',  'uint8',  true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'max', 'integer','integer',true, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
        100, 2, 'mod', 'double', 'double', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'mod', 'single', 'single', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'mod', 'int32',  'int32',  false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'mod', 'int16',  'int16',  false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'mod', 'int8',   'int8',   false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'mod', 'uint32', 'uint32', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'mod', 'uint16', 'uint16', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'mod', 'uint8',  'uint8',  false, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
        100, 2, 'rem', 'double', 'double', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'rem', 'single', 'single', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'rem', 'int32',  'int32',  false, 'UNSPECIFIED');
```

```
registerCustomizationEntry(hTable, ...
        100, 2, 'rem', 'int16',  'int16',  false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'rem', 'int8',   'int8',   false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'rem', 'uint32', 'uint32', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'rem', 'uint16', 'uint16', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 2, 'rem', 'uint8',  'uint8',  false, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
        100, 1, 'round', 'double', 'double', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'round', 'single', 'single', false, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
        100, 3, 'saturate', 'double', 'double', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 3, 'saturate', 'single', 'single', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 3, 'saturate', 'int32',  'int32',  true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 3, 'saturate', 'int16',  'int16',  true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 3, 'saturate', 'int8',   'int8',   true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 3, 'saturate', 'uint32', 'uint32', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 3, 'saturate', 'uint16', 'uint16', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 3, 'saturate', 'uint8',  'uint8',  true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 3, 'saturate', 'integer','integer',true, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
        100, 1, 'sign', 'double', 'double',  true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'sign', 'single', 'single',  true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'sign', 'int32',  'integer', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'sign', 'int16',  'integer', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'sign', 'int8',   'integer', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'sign', 'uint32', 'uint32',  true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'sign', 'uint16', 'uint16',  true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'sign', 'uint8',  'uint8',   true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'sign', 'integer','integer', true, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
```

```
        100, 1, 'sqrt', 'double', 'double', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'sqrt', 'single', 'single', true, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
        100, 1, 'fix', 'double', 'double', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'fix', 'single', 'single', false, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
        100, 1, 'copysign', 'double', 'double', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
        100, 1, 'copysign', 'single', 'single', true, 'UNSPECIFIED');
```

2. To reduce the size of the file, you can delete the `registerCustomizationEntry` lines for functions for which the default nonfinite number checking and inlining behavior is acceptable.
3. For each remaining entry,

   - Set the `Inline` argument to `true` if the function should be inlined or `false` if it should not be inlined.
   - Set the `SNF` argument to `ENABLE` if nonfinite checking should be generated, `DISABLE` if nonfinite checking should not be generated, or `UNSPECIFIED` to accept the default behavior based on the model option settings.

   Save the file.
4. Optionally, perform a quick check of the syntactic validity of the customization table entries by invoking the table definition file at the MATLAB command line (`>> tbl = crl_table_customization`). Fix syntax errors that are flagged.
5. Optionally, view the customization table entries in the Code Replacement Viewer (`>> crviewer(crl_table_customization)`). For more information about viewing code replacement tables, see "Choose a Code Replacement Library" (Simulink Coder).
6. To register these changes and make them appear in the **Code replacement library** drop-down list located on the **Code Generation** > **Interface** pane of the Configuration Parameters dialog box, first copy the following MATLAB function code into an instance of the file `rtwTargetInfo.m`.

Note: For the example below, specify the argument `'RTW'` if a GRT target is selected for your model, otherwise omit the argument.

```
function rtwTargetInfo(cm)
```

```
% rtwTargetInfo function to register a code replacement library (CRL)

  % Register the CRL defined in local function locCrlRegFcn
  cm.registerTargetInfo(@locCrlRegFcn);

end % End of RTWTARGETINFO


% Local function to define a CRL containing crl_table_customization
function thisCrl = locCrlRegFcn

  % Instantiate a CRL registry entry - specify 'RTW' for GRT
  thisCrl = RTW.TflRegistry('RTW');

  % Define the CRL properties
  thisCrl.Name = 'CRL Customization Example';
  thisCrl.Description = 'Example  of CRL Customization';
  thisCrl.TableList = {'crl_table_customization'};
  thisCrl.TargetHWDeviceType = {'*'};

end % End of LOCCRLREGFCN
```

You can edit the `Name` field to specify the library name that appears in the **Code replacement library** drop-down list. Also, the file name in the `TableList` field must match the name of the file you created in step 1.

To register your changes, with both of the MATLAB files you created present in the MATLAB path, enter the following command at the MATLAB command line:

```
sl_refresh_customizations
```

7. Create or open a model that generates function code corresponding to one of the math functions for which you specified a change in nonfinite number checking or inlining behavior.
8. Open the Configuration Parameters dialog box, go to the **Code Generation > Interface** pane, and use the **Code replacement library** drop-down list to select the code replacement entry you registered in step 6, for example, `CRL Customization Example`.
9. Generate code for the model and examine the generated code to verify that the math functions are generated as expected.

Minimize Computations and Storage for Intermediate Results at Block Outputs

In this section...

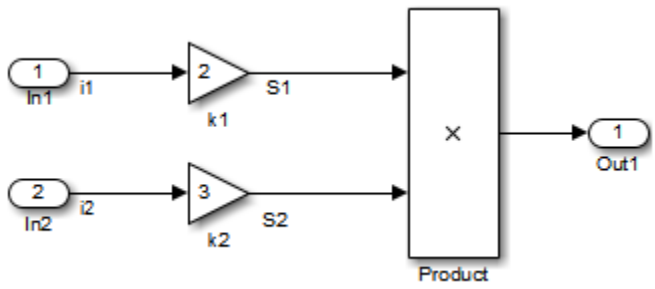
Expression Folding

Expression folding optimizes code to minimize the computation of intermediate results at block outputs and the storage of such results in temporary buffers or variables. When expression folding is on, the code generator collapses (folds) block computations into a single expression, instead of generating separate code statements and storage declarations for each block in the model. Most Simulink blocks support expression folding.

Expression folding improves the efficiency of generated code, frequently achieving results that compare favorably to hand-optimized code. In many cases, entire groups of model computations fold into a single, highly optimized line of code.

You can use expression folding in your own inlined S-function blocks. For more information, see "S-Functions That Support Expression Folding" (Simulink Coder).

Example Model



Generate Code

With expression folding off, in the `explfld.c` file, the code generator generates this code.

```
/* Model step function */
void exprfld_step(void)
{
  /* Gain: '<Root>/Gain' incorporates:
   *  Inport: '<Root>/In1'
   */
  exprfld_B.S1 = exprfld_P.Gain_Gain * exprfld_U.i1;

  /* Gain: '<Root>/Gain1' incorporates:
   *  Inport: '<Root>/In2'
   */
  exprfld_B.S2 = exprfld_P.Gain1_Gain * exprfld_U.i2;

  /* Outport: '<Root>/Out1' incorporates:
   *  Product: '<Root>/Product'
   */
  exprfld_Y.Out1 = exprfld_B.S1 * exprfld_B.S2;
}
```

There are separate code statements for both Gain blocks. Before final output, these code statements compute temporary results for the Gain blocks.

Enable Optimization

Expression folding is on by default. To see if expression folding is on for an existing model:

1. Open the Configuration Parameters dialog box and select the **All Parameters** tab.
2. Expression folding is available only when the **Signal storage reuse** parameter is set to `on` because expression folding operates only on expressions involving local variables. On the **All Parameters** tab, select **Signal storage reuse**.
3. When you select **Signal storage reuse**, the **Enable local block outputs**, **Reuse local block outputs**, and **Eliminate superfluous local variables (expression folding)** parameters are all on by default.

Generate Code with Optimization

With expression folding, the code generator generates a single-line output computation, as shown in the `expfld.c` file. The generated comments document the block parameters that appear in the expression.

```
/* Model step function */
void exprfld_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Gain: '<Root>/Gain'
   *  Gain: '<Root>/Gain1'
   *  Inport: '<Root>/In1'
   *  Inport: '<Root>/In2'
   *  Product: '<Root>/Product'
   */
  exprfld_Y.Out1 = exprfld_P.Gain_Gain * exprfld_U.i1 * (exprfld_P.Gain1_Gain *
    exprfld_U.i2);
}
```

For an example of expression folding in the context of a more complex model, click `rtwdemo_slexprfold` , or at the command prompt, type:

```
rtwdemo_slexprfold
```

For more information, see “Enable and Reuse Local Block Outputs in Generated Code” (Simulink Coder)

See Also

“Signal storage reuse” (Simulink) | “Reuse local block outputs” (Simulink) | “Enable local block outputs” (Simulink) | “Eliminate superfluous local variables (Expression folding)” (Simulink)

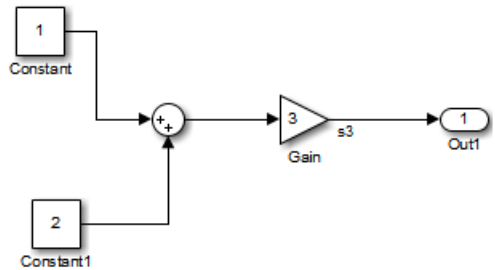
Related Examples

- “Optimization Tools and Techniques” on page 53-7
- “Control Signals and States in Code by Applying Storage Classes” on page 19-123

Inline Invariant Signals

You can optimize the generated code by selecting **Inline invariant signals** on the **Optimization** > **Signals and Parameters** pane. The generated code uses the numerical values of the invariant signals instead of their symbolic names.

An invariant signal is a block output signal that does not change during Simulink simulation. For example, the signal S3 is an invariant signal. An *invariant signal* is not the same as an *invariant constant*. The two constants (1 and 2) and the gain value of 3 are invariant constants. To inline invariant constants, set **Default parameter behavior** to `Inlined`.



Optimize Generated Code Using Inline Invariant Signals

This example shows how to use inline invariant signals to optimize the generated code. This optimization transforms symbolic names of invariant signals into constant values.

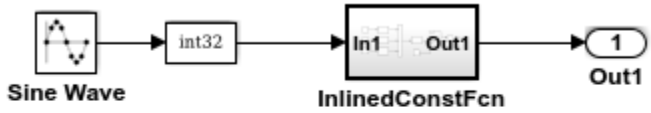
The `InlineInvariantSignals` optimization:

- Reduces ROM and RAM consumption.
- Improves execution speed.

Example Model

Consider the model matlab:rtwdemo_inline_invariant_signals.

```
model = 'rtwdemo_inline_invariant_signals';
open_system(model);
```



Generate Code

Create a temporary folder (in your system temporary folder) for the build and inspection process.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
```

Build the model using Simulink Coder.

```
rtwbuild(model)
```

```
### Starting build procedure for model: rtwdemo_inline_invariant_signals
### Successful completion of build procedure for model: rtwdemo_inline_invariant_signal
```

View the generated code without the optimization. These lines of code are in `rtwdemo_inline_invariant_signals.c`.

```
cfile = fullfile(cgDir,'rtwdemo_inline_invariant_signals_grt_rtw',...
    'rtwdemo_inline_invariant_signals.c');
rtwdemodbtype(cfile,'/* Output and update for atomic system',...
    '/* Model output', 1, 0);
```

```
/* Output and update for atomic system: '<Root>/InlinedConstFcn' */
void rtwdemo_inline_InlinedConstFcn(int32_T rtu_In1,
  B_InlinedConstFcn_rtwdemo_inl_T *localB, const ConstB_InlinedConstFcn_rtwdem_T
  *localC)
{
  /* Product: '<S1>/Product' */
  localB->Product = rtu_In1 * localC->Sum_p;
}
```

Enable Optimization

1. Open the Configuration Parameters dialog box.
2. On the **Optimization->Signals and Parameters** pane, select **Inline Invariant Signals**.

Alternatively, you can use the command-line API to enable the optimization:

```
set_param(model, 'InlineInvariantSignals', 'on');
```

Generate Code with Optimization

The generated code uses the numerical values of the folded constants instead of creating an additional structure (rtwdemo_inline_invariant_ConstB).

Build the model using Simulink Coder.

```
rtwbuild(model)
```

```
### Starting build procedure for model: rtwdemo_inline_invariant_signals
### Successful completion of build procedure for model: rtwdemo_inline_invariant_signal
```

View the generated code with the optimization. These lines of code are in `rtwdemo_minmax.c`.

```
rtwdemodbtype(cfile,'/* Output and update for atomic system', '/* Model output', 1, 0);
```

```
/* Output and update for atomic system: '<Root>/InlinedConstFcn' */
void rtwdemo_inline__InlinedConstFcn(int32_T rtu_In1,
  B_InlinedConstFcn_rtwdemo_inl_T *localB)
{
  /* Product: '<S1>/Product' */
  localB->Product = rtu_In1 << 5;
}
```

Close the model and code generation report.

```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

See Also

"Inline invariant signals" (Simulink)

Related Examples

- “Optimization Tools and Techniques” on page 53-7
- “Inline Numeric Values of Block Parameters” on page 53-43

Inline Numeric Values of Block Parameters

This example shows how to optimize the generated code by inlining the numeric values of block parameters. *Block parameters* include the **Gain** parameter of a Gain block and the table data and breakpoint sets of an n-D Lookup Table block.

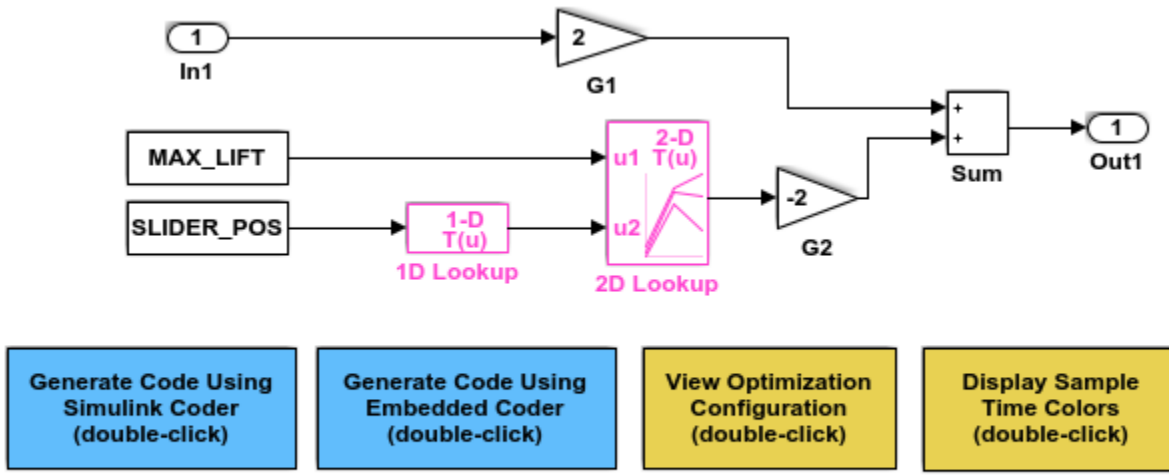
This optimization determines whether numeric block parameters occupy global memory in the generated code. The optimization can:

- Improve execution speed.
- Reduce RAM and ROM consumption.

Explore Example Model

Open the example model `rtwdemo_paraminline`.

```
open_system('rtwdemo_paraminline')
```



The model contains blocks that have these numeric parameters:

- The **Gain** parameters of the Gain blocks

- The **Constant value** parameters of the Constant blocks
- The table data and breakpoint sets of the n-D Lookup Table blocks

The output of the block G2, and the outputs of blocks upstream of G2, change only if you tune the values of the block parameters during simulation or during code execution. When you update the model diagram, these blocks and signal lines appear magenta in color.

Several blocks use `Simulink.Parameter` objects in the base workspace to set the values of their parameters. The parameter objects all use the storage class `Auto`, which means that you can configure the generated code to inline the parameter values.

Generate Code Without Optimization

Create a temporary folder for the build and inspection process.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
```

Disable the optimization by setting **Configuration Parameters > Optimization > Signals and Parameters > Default parameter behavior** to `Tunable`.

```
set_param('rtwdemo_paraminline','DefaultParameterBehavior','Tunable')
```

Generate code from the model.

```
rtwbuild('rtwdemo_paraminline')
```

```
### Starting build procedure for model: rtwdemo_paraminline
### Successful completion of build procedure for model: rtwdemo_paraminline
```

In the code generation report, view the source file `rtwdemo_paraminline_data.c`. The code defines a global structure that contains the block parameter values. Each block parameter in the model, such as a lookup table array, breakpoint set, or gain, appears as a field of the structure.

```
cfile = fullfile(cgDir,'rtwdemo_paraminline_grt_rtw','rtwdemo_paraminline_data.c');
rtwdemodbtype(cfile,'/* Block parameters (auto storage) */', '};', 1, 1);
```

```
/* Block parameters (auto storage) */
P_rtwdemo_paraminline_T rtwdemo_paraminline_P = {
  10.0,                                /* Variable: MAX_LIFT
                                        * Referenced by: '<Root>/Constant'
                                        */
```

```
  0.0,                                 /* Variable: SLIDER_POS
                                        * Referenced by: '<Root>/Constant1'
                                        */

  /*  Variable: T1Break
   * Referenced by: '<Root>/1D Lookup'
   */
  { -5.0, -4.0, -3.0, -2.0, -1.0, 0.0, 1.0, 2.0, 3.0, 4.0, 5.0 },

  /*  Variable: T1Data
   * Referenced by: '<Root>/1D Lookup'
   */
  { -1.0, -0.99, -0.98, -0.96, -0.76, 0.0, 0.76, 0.96, 0.98, 0.99, 1.0 },

  /*  Variable: T2Break
   * Referenced by: '<Root>/2D Lookup'
   */
  { 1.0, 2.0, 3.0 },

  /*  Variable: T2Data
   * Referenced by: '<Root>/2D Lookup'
   */
  { 4.0, 16.0, 10.0, 5.0, 19.0, 18.0, 6.0, 20.0, 23.0 },
  2.0,                                 /* Expression: 2
                                        * Referenced by: '<Root>/G1'
                                        */
  -2.0,                                /* Expression: -2
                                        * Referenced by: '<Root>/G2'
                                        */

  /*  Computed Parameter: uDLookup_maxIndex
   * Referenced by: '<Root>/2D Lookup'
   */
  { 2U, 2U }
};
```

You can tune the structure fields during code execution because they occupy global memory. However, at each step of the generated algorithm, the code must calculate the output of each block, including the outputs of the block G2 and the upstream blocks. View the algorithm in the model `step` function in the file `rtwdemo_paraminline.c`.

```
cfile = fullfile(cgDir,'rtwdemo_paraminline_grt_rtw','rtwdemo_paraminline.c');
rtwdemodbtype(cfile,'/* Model step function */','/* Model initialize function */',1,0);
```

```
/* Model step function */
void rtwdemo_paraminline_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Constant: '<Root>/Constant'
   *  Constant: '<Root>/Constant1'
   *  Gain: '<Root>/G1'
   *  Gain: '<Root>/G2'
   *  Inport: '<Root>/In1'
   *  Lookup_n-D: '<Root>/1D Lookup'
   *  Lookup_n-D: '<Root>/2D Lookup'
   *  Sum: '<Root>/Sum'
   */
  rtwdemo_paraminline_Y.Out1 = rtwdemo_paraminline_P.G1_Gain *
    rtwdemo_paraminline_U.In1 + rtwdemo_paraminline_P.G2_Gain * look2_binlx
    (rtwdemo_paraminline_P.MAX_LIFT, look1_binlx
     (rtwdemo_paraminline_P.SLIDER_POS, rtwdemo_paraminline_P.T1Break,
      rtwdemo_paraminline_P.T1Data, 10U), rtwdemo_paraminline_P.T2Break,
     rtwdemo_paraminline_P.T2Break, rtwdemo_paraminline_P.T2Data,
     rtwdemo_paraminline_P.uDLookup_maxIndex, 3U);
}
```

Generate Code with Optimization

Set **Default parameter behavior** to `Inlined`.

```
set_param('rtwdemo_paraminline','DefaultParameterBehavior','Inlined')
```

Generate code from the model.

```
rtwbuild('rtwdemo_paraminline')
```

```
### Starting build procedure for model: rtwdemo_paraminline
### Successful completion of build procedure for model: rtwdemo_paraminline
```

In the code generation report, view the algorithm in the file `rtwdemo_paraminline.c`.

```
rtwdemodbtype(cfile,'/* Model step function */','/* Model initialize function */',1,0)
```

```
/* Model step function */
void rtwdemo_paraminline_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Gain: '<Root>/G1'
```

```
   *  Inport: '<Root>/In1'
   *  Sum: '<Root>/Sum'
   */
  rtwdemo_paraminline_Y.Out1 = 2.0 * rtwdemo_paraminline_U.In1 + 150.0;
}
```

The code does not allocate memory for block parameters or for parameter objects that use the storage class `Auto`. Instead, the code generator uses the parameter values from the model, and from the parameter objects, to calculate and inline the constant output of the block G2, `150.0`. The generator also inlines the value of the **Gain** parameter of the Gain block G1, `2.0`.

With the optimization, the generated code leaves out computationally expensive algorithmic code for blocks such as the lookup tables. The optimized code calculates the output of a block only if the output can change during execution. For this model, only the outputs of the Inport block In1, the Gain block G1, and the Sum block can change.

Close the model and the code generation report.

```
bdclose('rtwdemo_paraminline')
rtwdemoclean;
cd(currentDir)
```

Preserve Block Parameter Tunability

When you set **Default parameter behavior** to `Inlined`, you can preserve block parameter tunability by creating `Simulink.Parameter` objects for individual parameters. You can configure each object to appear in the code as a tunable field of the global parameter structure or as an individual global variable. You can change parameter values during code execution and interface the generated code with your own handwritten code. For more information, see “Block Parameter Representation in the Generated Code” (Simulink Coder).

Inline Invariant Signals

You can select the **Inline invariant signals** code generation option (which also places constant values in the generated code) only when you set **Default parameter behavior** to `Inlined`. See “Inline Invariant Signals” (Simulink Coder).

See Also

“Default parameter behavior” (Simulink)

Related Examples

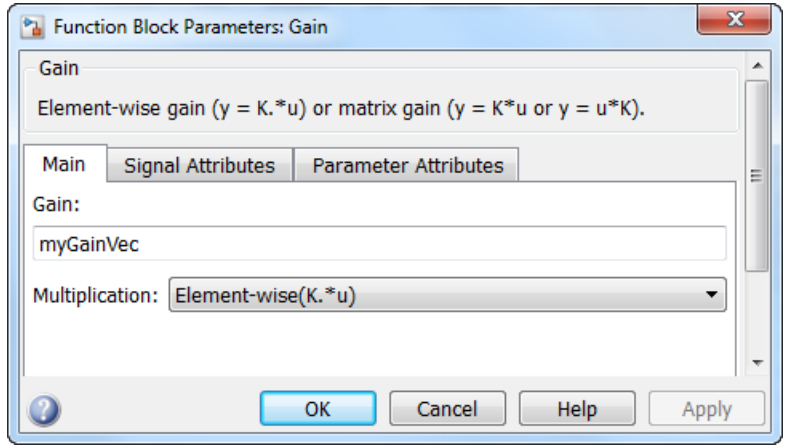
- “Block Parameter Representation in the Generated Code” (Simulink Coder)

Configure Loop Unrolling Threshold

The **Loop unrolling threshold** parameter on the **Optimization > Signals and Parameters** pane determines when a wide signal or parameter should be wrapped into a `for` loop and when it should be generated as a separate statement for each element of the signal. The default threshold value is 5.

For example, consider the model below:

The gain parameter of the Gain block is the vector `myGainVec`.



Assume that the loop unrolling threshold value is set to the default, 5.

If `myGainVec` is declared as

```
myGainVec = [1:10];
```

an array of 10 elements, `myGainVec_P.Gain_Gain[]`, is declared within the `Parameters_`*`model`* data structure. The size of the gain array exceeds the loop unrolling threshold. Therefore, the code generated for the Gain block iterates over the array in a `for` loop, as shown in the following code:

```
{
    int32_T i1;

    /* Gain: '<Root>/Gain' */
    for(i1=0; i1<10; i1++) {
      myGainVec_B.Gain_f[i1] = rtb_foo *
        myGainVec_P.Gain_Gain[i1];
    }
  }
```

If `myGainVec` is declared as

```
myGainVec = [1:3];
```

an array of three elements, `myGainVec_P.Gain_Gain[]`, is declared within the `Parameters` data structure. The size of the gain array is below the loop unrolling threshold. The generated code consists of inline references to each element of the array, as in the code below.

```
/* Gain: '<Root>/Gain' */
  myGainVec_B.Gain_f[0] = rtb_foo * myGainVec_P.Gain_Gain[0];
  myGainVec_B.Gain_f[1] = rtb_foo * myGainVec_P.Gain_Gain[1];
  myGainVec_B.Gain_f[2] = rtb_foo * myGainVec_P.Gain_Gain[2];
```

See “Explore Variable Names and Loop Rolling” (Simulink Coder) for more information on loop rolling.

Note When a model includes Stateflow charts or MATLAB Function blocks, you can apply a set of Stateflow optimizations on the **Optimization > Stateflow** pane. The settings you select for the Stateflow options also apply to MATLAB Function blocks in the model. This is because the MATLAB Function blocks and Stateflow charts are built on top of the same technology and share a code base. You do not need a Stateflow license to use MATLAB Function blocks.

See Also

“Loop unrolling threshold” (Simulink)

Related Examples

- “Optimization Tools and Techniques” on page 53-7
- “Optimize Generated Code by Combining Multiple `for` Constructs” on page 53-15

- “For Loop” on page 13-40

Use `memcpy` Function to Optimize Generated Code for Vector Assignments

In this section...

You can use the **Use memcpy for vector assignment** parameter to optimize generated code for vector assignments by replacing `for` loops with `memcpy` function calls. The `memcpy` function is more efficient than `for`-loop controlled element assignment for large data sets. This optimization improves execution speed.

Selecting the **Use memcpy for vector assignment** parameter enables the associated parameter **Memcpy threshold (bytes)**, which allows you to specify the minimum array size in bytes for which `memcpy` function calls should replace `for` loops in the generated code. For more information, see "Use memcpy for vector assignment" (Simulink) and "Memcpy threshold (bytes)" (Simulink). In considering whether to use this optimization,

- Verify that your target supports the `memcpy` function.
- Determine whether your model uses signal vector assignments (such as `Y=expression`) to move large amounts of data, for example, using the Selector block.

To apply this optimization,

1. Consider first generating code without this optimization and measuring its execution speed, to establish a baseline for evaluating the optimized assignment.
2. Select **Use memcpy for vector assignment** and examine the setting of **Memcpy threshold (bytes)**, which by default specifies 64 bytes as the minimum array size for which `memcpy` function calls replace `for` loops. Based on the array sizes used in your application's signal vector assignments, and target environment considerations that might bear on the threshold selection, accept the default or specify another array size.
3. Generate code, and measure its execution speed against your baseline or previous iterations. Iterate on steps 2 and 3 until you achieve an optimal result.

Note: The `memcpy` optimization may not occur under certain conditions, including when other optimizations have a higher precedence than the `memcpy` optimization, or when the

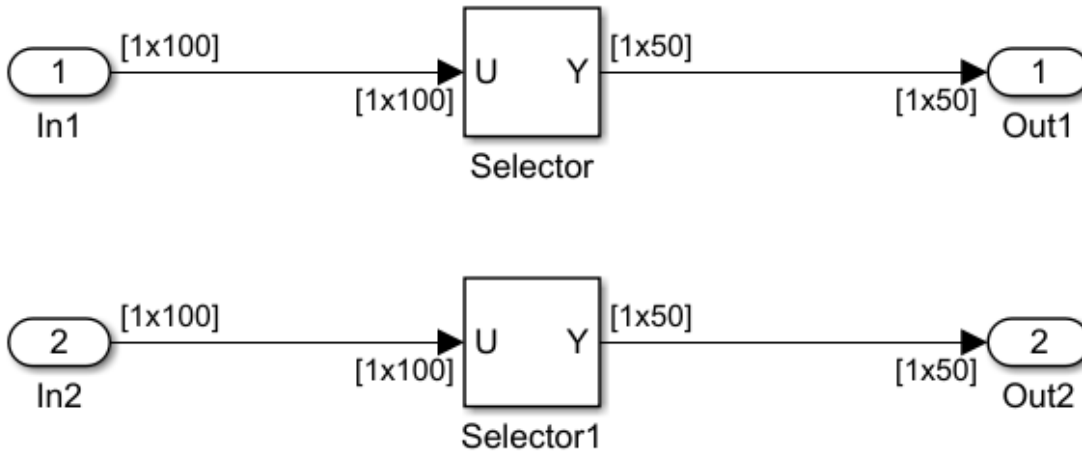
generated code is originating from Target Language Compiler (TLC) code, such as a TLC file associated with an S-function block.

Note: If you are licensed for Embedded Coder software, you can use a code replacement library (CRL) to provide your own custom implementation of the `memcpy` function to be used in generated model code. For more information, see "Memory Function Code Replacement" on page 51-96.

Example Model

To examine the result of using the **Use memcpy for vector assignment** parameter on the generated vector assignment code, create a model that generates signal vector assignments. For example,

1. Use In, Out, and Selector blocks to create the following model.



2. Open Model Explorer and configure the **Signal Attributes** for the `In1` and `In2` source blocks. For each, set **Port dimensions** to `[1,100]`, and set **Data type** to `int32`. Apply the changes and save the model. In this example, the model has the name `vectorassign`.
3. For each Selector block, set the **Index** parameter to `1:50`. Set the **Input port size** parameter to `100`.

Generate Code

1. The **Use memcpy for vector assignment** parameter is on by default. To turn off the parameter, go to the **Optimization > Signals and Parameters** pane and clear the **Use memcpy for vector assignment** parameter.
2. Go to the **Code Generation > Report** pane of the Configuration Parameters dialog box and select the **Create code generation report**. Then go to the **Code Generation** pane, select the **Generate code only** option, and generate code for the model. When code generation completes, the HTML code generation report is displayed.
3. In the HTML code generation report, click the `vectorassign.c` section and inspect the model step function. Notice that the vector assignments are implemented using `for` loops.

```
/* Model step function */
void vectorassign_step(void)
{
  int32_T i;
  for (i = 0; i < 50; i++) {
    /* Outport: '<Root>/Out1' incorporates:
     *  Inport: '<Root>/In1'
     */
    vectorassign_Y.Out1[i] = vectorassign_U.In1[i];

    /* Outport: '<Root>/Out2' incorporates:
     *  Inport: '<Root>/In2'
     */
    vectorassign_Y.Out2[i] = vectorassign_U.In2[i];
  }
}
```

Generate Code with Optimization

1. Go to the **Optimization > Signals and Parameters** pane of the Configuration Parameters dialog box and select the **Use memcpy for vector assignment** option. Leave the **Memcpy threshold (bytes)** option at its default setting of `64`. Apply the changes and regenerate code for the model. When code generation completes, the HTML code generation report again is displayed.
2. In the HTML code generation report, click the `vectorassign.c` section and inspect the model output function. Notice that the vector assignments now are implemented using `memcpy` function calls.

```
/* Model step function */
void vectorassign_step(void)
{
```

```
  /* Outport: '<Root>/Out1' incorporates:
   *  Inport: '<Root>/In1'
   */
  memcpy(&vectorassign_Y.Out1[0], &vectorassign_U.In1[0], 50U * sizeof(real_T));

  /* Outport: '<Root>/Out2' incorporates:
   *  Inport: '<Root>/In2'
   */
  memcpy(&vectorassign_Y.Out2[0], &vectorassign_U.In2[0], 50U * sizeof(real_T));
}
```

See Also

"Use memcpy for vector assignment" (Simulink)

Related Examples

- "Optimization Tools and Techniques" on page 53-7
- "Vector Operation Optimization" on page 53-97
- "Convert Data Copies to Pointer Assignments" on page 57-23

Generate Target Optimizations Within Algorithm Code

Some application components are hardware-specific and cannot simulate on a host system. For example, consider a component that includes `pragmas` and assembly code for enabling hardware instructions for saturate on add operations or a Fast Fourier Transform (FFT) function.

The following table lists integration options to customize generated algorithm code with target-specific optimizations.

Note: Solutions marked with *EC only* require an Embedded Coder license.

If...	Then...	For More Information, See
You want to optimize the execution speed and memory of the model code by replacing default math functions and operators with target-specific code	*EC only*—Implement function and operator replacements by using the Code Replacement Tool, code replacement library (CRL) API, and Code Replacement Viewer to create, examine, validate, and register hardware-specific replacement tables	"Optimize Generated Code By Developing and Using Code Replacement Libraries - Simulink®"
You want to control how code generation technology declares, stores, and represents signals, tunable parameters, block states, and data objects in generated code	*EC only*—Design (create) and apply custom storage classes	• `rtwdemo_cscpredef` • `rtwdemo_importstruct` • `rtwdemo_advsc` • "Custom Storage Classes"

Note: To simulate an algorithm that includes target-specific elements in a host environment, you must create code that is equivalent to the target code and can run in the host environment.

Remove Code for Blocks That Have No Effect on Computational Results

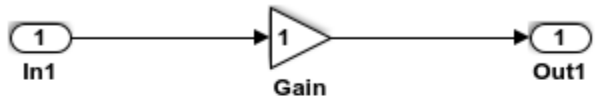
This example shows how the code generator optimizes generated code by removing code that has no effect on computational results. This optimization:

- Increases execution speed.
- Reduces ROM consumption.

Example

In the model rtwdemo_blockreduction, a Gain block of value `1.0` is in between Inport and Outport blocks.

```
model = 'rtwdemo_blockreduction';
open_system(model);
```



Generate Code

Create a temporary folder for the build and inspection process.

```
currentDir=pwd;
[~,cgDir]=rtwdemodir();
```

Build the model.

```
set_param(model,'BlockReduction','off');
rtwbuild(model)
```

```
### Starting build procedure for model: rtwdemo_blockreduction
### Successful completion of build procedure for model: rtwdemo_blockreduction
```

Here is the code from `rtwdemo_blockreduction.c`.

```
cfile = fullfile(cgDir,'rtwdemo_blockreduction_ert_rtw','rtwdemo_blockreduction.c');
rtwdemodbtype(cfile, '/* Model step function */',...
    '/* Model initialize function */', 1, 0);
```

```
/* Model step function */
void rtwdemo_blockreduction_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Gain: '<Root>/Gain'
   *  Inport: '<Root>/In1'
   */
  rtwdemo_blockreduction_Y.Out1 = 1.0 * rtwdemo_blockreduction_U.In1;
}
```

Enable Optimization

1. Open the Configuration Parameters dialog box.
2. On the **All Parameters** tab, select **Block reduction**. This optimization is on by default.

Alternately, use the command-line API to enable the optimization.

```
set_param(model,'BlockReduction','on');
```

Generate Code with Optimization

```
rtwbuild(model)
```

```
### Starting build procedure for model: rtwdemo_blockreduction
### Successful completion of build procedure for model: rtwdemo_blockreduction
```

Here is the optimized code from `rtwdemo_blockreduction.c`.

```
cfile = fullfile(cgDir,'rtwdemo_blockreduction_ert_rtw','rtwdemo_blockreduction.c');
rtwdemodbtype(cfile, '/* Model step function */',...
    '/* Model initialize function */', 1, 0);
```

```
/* Model step function */
void rtwdemo_blockreduction_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
```

```
   *  Inport: '<Root>/In1'
   */
  rtwdemo_blockreduction_Y.Out1 = rtwdemo_blockreduction_U.In1;
}
```

Because multiplying the input signal by a value of `1.0` does not impact computational results, the code generator excludes the Gain block from the generated code. Close the model and clean up.

```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

See Also

"Block reduction" (Simulink)

Related Examples

- "Optimization Tools and Techniques" on page 53-7
- "Inline Numeric Values of Block Parameters" on page 53-43
- "Minimize Computations and Storage for Intermediate Results at Block Outputs" on page 53-36

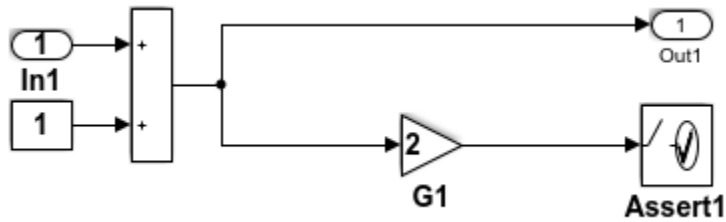
Eliminate Dead Code Paths in Generated Code

This example shows how the code generator eliminates dead (that is, unused) code paths from generated code. This optimization increases execution speed and conserves ROM and RAM consumption.

Example

In the model rtwdemo_deadpathElim, the signal leaving the Sum block divides into two separate code paths. The top path is not a dead code path. If the user disables the Assertion block, the bottom path becomes a dead code path.

```
model = 'rtwdemo_deadpathElim';
open_system(model);
```



Generate Code with an Enabled Assertion Block

1. For the Assertion block, open the block parameters dialog box.
2. Select the **Enable assertion** box. Alternatively, use the command-line API to enable the Assertion block.

```
set_param([model '/Assert1'], 'Enabled', 'on');
```

Create a temporary folder for the build and inspection process.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
```

Build the model.

```
rtwbuild(model)
```

```
### Starting build procedure for model: rtwdemo_deadpathElim
```

```
### Successful completion of build procedure for model: rtwdemo_deadpathElim
```

Because the Assertion block is enabled, these lines of `rtwdemo_deadpathElim.c` include code for the Gain and Assertion blocks.

```
cfile = fullfile(cgDir,'rtwdemo_deadpathElim_grt_rtw','rtwdemo_deadpathElim.c');
rtwdemodbtype(cfile,'/* Model step', '/* Model initialize function */', 0, 1);
```

```
void rtwdemo_deadpathElim_step(void)
{
  /* Sum: '<Root>/Sum1' incorporates:
   *  Constant: '<Root>/Constant1'
   *  Inport: '<Root>/In1'
   */
  rtwdemo_deadpathElim_Y.Out1 = rtwdemo_deadpathElim_U.In1 + 1.0;

  /* Assertion: '<Root>/Assert1' incorporates:
   *  Gain: '<Root>/G1'
   */
  utAssert(2.0 * rtwdemo_deadpathElim_Y.Out1 != 0.0);
}
```

Generate Code with a Disabled Assertion Block

Disable the Assertion block to generate a dead code path. The code generator detects the dead code path and eliminates it from the generated code.

1. For the Assertion block, open the Block Parameters dialog box.
2. Deselect the **Enable assertion** box.

Alternatively, use the command-line API to disable the Assertion block.

```
set_param([model '/Assert1'], 'Enabled', 'off');
```

Build the model.

```
rtwbuild(model)
```

```
### Starting build procedure for model: rtwdemo_deadpathElim
### Successful completion of build procedure for model: rtwdemo_deadpathElim
```

Because the Assertion block is disabled, these lines of `rtwdemo_deadpathElim.c` do not include code for the Gain and Assertion blocks.

```
rtwdemodbtype(cfile,'/* Model step', '/* Model initialize function */', 0, 1);
```

```
void rtwdemo_deadpathElim_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Constant: '<Root>/Constant1'
   *  Inport: '<Root>/In1'
   *  Sum: '<Root>/Sum1'
   */
  rtwdemo_deadpathElim_Y.Out1 = rtwdemo_deadpathElim_U.In1 + 1.0;
}
```

Close the model and clean-up.

```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

For another example of how the code generator eliminates dead code paths in the generated code, see rtwdemo_deadpath.

Related Examples

- “Optimization Tools and Techniques” on page 53-7
- “Remove Code for Blocks That Have No Effect on Computational Results” on page 53-58

Floating-Point Multiplication to Handle a Net Slope Correction

This example shows how to use floating-point multiplication to handle a net slope correction. When converting floating-point data types to fixed-point data types in the generated code, a net slope correction is one method of scaling fixed-point data types. Scaling the fixed-point data types avoids overflow conditions and minimizes quantization errors.

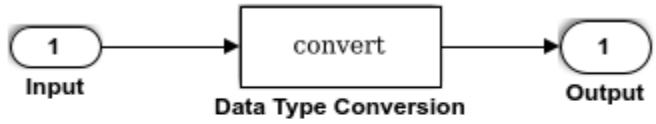
For processors that support efficient multiplication, using floating-point multiplication to handle a net slope correction improves code efficiency. If the net slope correction has a value that is not a power of two, using division improves precision.

Note: This example requires a Fixed-Point Designer™ license.

Example

In the model rtwdemo_float_mul_for_net_slope_correction, a Convert block converts an input signal from a floating-point data type to a fixed-point data type. The net slope correction has a value of 3.

```
model = 'rtwdemo_float_mul_for_net_slope_correction';
open_system(model);
```



Generate Code

Create a temporary folder for the build and inspection process.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
```

Build the model.

```
rtwbuild(model)

### Starting build procedure for model: rtwdemo_float_mul_for_net_slope_correction
### Successful completion of build procedure for model: rtwdemo_float_mul_for_net_slope
```

In these lines of `rtwdemo_float_mul_for_net_slope_correction.c` code, the code generator divides the input signal by `3.0F` .

```
cfile = fullfile(cgDir,'rtwdemo_float_mul_for_net_slope_correction_ert_rtw',...
    'rtwdemo_float_mul_for_net_slope_correction.c');
rtwdemodbtype(cfile,'/* Model step', '/* Model initialize', 1, 0);


/* Model step function */
void rtwdemo_float_mul_for_net_slope_correction_step(void)
{
  /* Outport: '<Root>/Output' incorporates:
   *  DataTypeConversion: '<Root>/Data Type Conversion'
   *  Inport: '<Root>/Input'
   */
  rtY.Output = (int16_T)(real32_T)floor((real_T)(rtU.Input / 3.0F));
}
```

Enable Optimization

1. Open the Configuration Parameters dialog box.
2. On the **Optimization** pane, select **Use floating-point multiplication to handle net slope corrections**. This optimization is on by default.

Alternatively, you can use the command-line API to enable the optimization.

```
set_param(model, 'UseFloatMulNetSlope', 'on');
```

Generate Code with Optimization

```
rtwbuild(model)

### Starting build procedure for model: rtwdemo_float_mul_for_net_slope_correction
### Successful completion of build procedure for model: rtwdemo_float_mul_for_net_slope
```

In the optimized code, the code generator multiplies the input signal by the reciprocal of `3.0F` , that is `0.333333343F` .

```
rtwdemodbtype(cfile,'/* Model step', '/* Model initialize', 1, 0);
```

```
/* Model step function */
void rtwdemo_float_mul_for_net_slope_correction_step(void)
{
  /* Outport: '<Root>/Output' incorporates:
   *  DataTypeConversion: '<Root>/Data Type Conversion'
   *  Inport: '<Root>/Input'
   */
  rtY.Output = (int16_T)(real32_T)floor((real_T)(rtU.Input * 0.333333343F));
}
```

Close the model and the code generation report.

```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

See Also

"Use floating-point multiplication to handle net slope corrections" (Simulink)

Related Examples

- "Optimization Tools and Techniques" on page 53-7
- "Remove Code From Floating-Point to Integer Conversions That Wraps Out-of-Range Values" on page 53-23
- "Subnormal Number Performance" on page 53-18

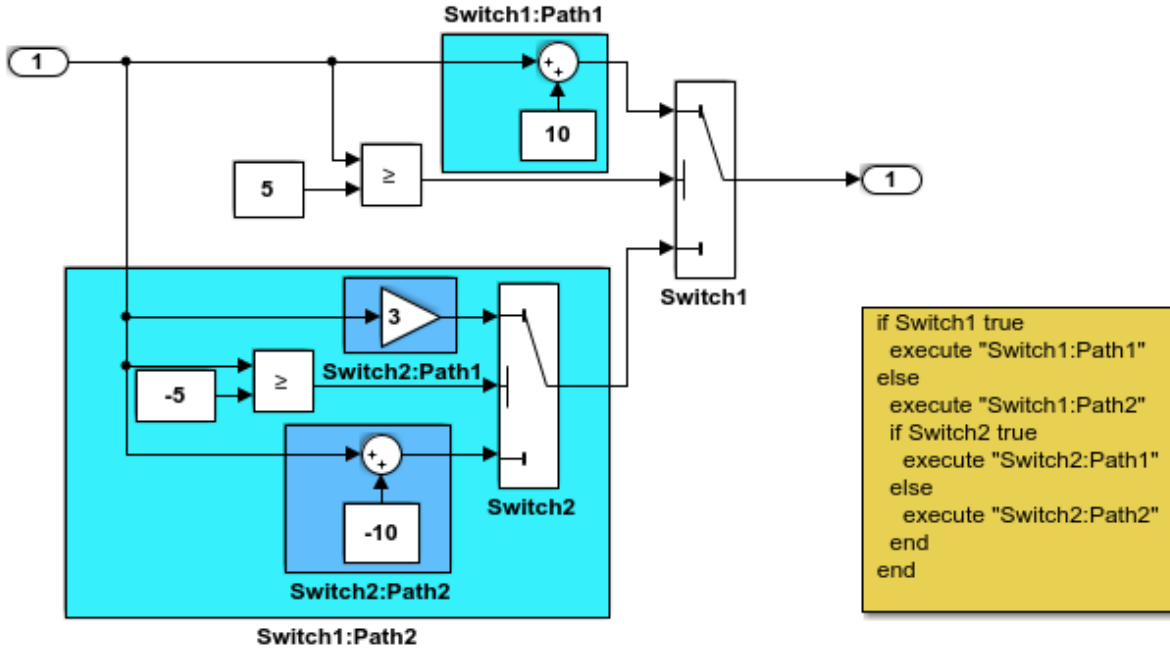
Use Conditional Input Branch Execution

This example shows how to optimize the generated code for a model that contains Switch and Multiport Switch blocks. When you select the model configuration parameter **Conditional input branch execution**, Simulink executes only blocks that compute the control input and data input that the control input selects. This optimization improves execution speed.

Example Model

In this example, switch paths are conditionally executed. If `Switch1` control input is true, `Switch1` executes blocks grouped in the `Switch1:Path1` branch. If `Switch1` control input is false, `Switch1` executes blocks grouped in the `Switch1:Path2` branch. If `Switch1` executes blocks in the `Switch1:Path2` branch and `Switch2` control input is true, `Switch2` executes blocks in the `Switch2:Path1` branch. If `Switch2` control input is false, `Switch2` executes blocks in the `Switch2:Path2` branch. The pseudo code shows this logic.

```
model='rtwdemo_condinput';
open_system(model);
```



This model shows conditional input branch execution performed by Simulink and Simulink Coder. Conditional input branch execution improves simulation and code generation execution performance. In this example, switch paths are conditionally executed. That is, if Switch1's control input is true, Switch1 executes blocks grouped in the "Switch1:Path1" branch. Otherwise, it executes blocks grouped in the "Switch1:Path2" branch. If blocks in "Switch1:Path2" are executed, Switch2 executes the "Switch2:Path1" branch if its control input is true. Otherwise, it executes the "Switch2:Path2" branch. This logic is illustrated by the pseudo code above.

Generate Code Using Simulink Coder (double-click)

Generate Code Using Embedded Coder (double-click)

Generate Code

The **Conditional input branch execution** parameter is on by default. Enter the following command-line API to turn off the parameter.

```
set_param(model, 'ConditionallyExecuteInputs', 'off');
```

Create a temporary folder for the build and inspection process.

```
currentDir=pwd;
[~,cgDir]=rtwdemodir();
```

Build the model.

```
rtwbuild(model)
```

```
### Starting build procedure for model: rtwdemo_condinput
### Successful completion of build procedure for model: rtwdemo_condinput
```

View the generated code without the optimization. These lines of code are in the `rtwdemo_condinput.c` file.

```
cfile = fullfile(cgDir,'rtwdemo_condinput_grt_rtw','rtwdemo_condinput.c');
rtwdemodbtype(cfile,'/* Model step', '/* Model initialize', 1, 0);
```

```
/* Model step function */
void rtwdemo_condinput_step(void)
{
  /* Switch: '<Root>/ Switch2' incorporates:
   *  Constant: '<Root>/ C_10'
   *  Constant: '<Root>/C_5'
   *  Gain: '<Root>/  G3'
   *  Inport: '<Root>/input'
   *  RelationalOperator: '<Root>/Relational Operator'
   *  Sum: '<Root>/ Sum'
   */
  if (rtwdemo_condinput_U.input >= -5.0) {
    rtwdemo_condinput_Y.output = 3.0 * rtwdemo_condinput_U.input;
  } else {
    rtwdemo_condinput_Y.output = rtwdemo_condinput_U.input + -10.0;
  }

  /* End of Switch: '<Root>/ Switch2' */
```

```
  /* Switch: '<Root>/Switch1' incorporates:
   *  Constant: '<Root>/C5'
   *  Inport: '<Root>/input'
   *  RelationalOperator: '<Root>/Relational Operator1'
   */
  if (rtwdemo_condinput_U.input >= 5.0) {
    /* Outport: '<Root>/output' incorporates:
     *  Constant: '<Root>/  C10'
     *  Sum: '<Root>/ Sum1'
     */
    rtwdemo_condinput_Y.output = rtwdemo_condinput_U.input + 10.0;
  }

  /* End of Switch: '<Root>/Switch1' */
}
```

The generated code contains an `if-else` statement for the `Switch1` block and an `if` statement for the `Switch2` block. Therefore, the generated code for `Switch1:Path2` executes even if the `if` statement for `Switch1:Path1` evaluates to true.

Enable Optimization

1. Open the Configuration Parameters dialog box.
2. On the **All Parameters** tab, select **Conditional input branch execution**. Alternatively, you can use the command-line API to enable the optimization.

```
set_param(model, 'ConditionallyExecuteInputs','on');
```

Generate Code with Optimization

```
rtwbuild(model)
cfile = fullfile(cgDir,'rtwdemo_condinput_grt_rtw','rtwdemo_condinput.c');
rtwdemodbtype(cfile,'/* Model step', '/* Model initialize', 1, 0);

### Starting build procedure for model: rtwdemo_condinput
### Successful completion of build procedure for model: rtwdemo_condinput

/* Model step function */
void rtwdemo_condinput_step(void)
{
  /* Switch: '<Root>/Switch1' incorporates:
   *  Constant: '<Root>/C5'
   *  Constant: '<Root>/C_5'
   *  Inport: '<Root>/input'
```

```
   *  RelationalOperator: '<Root>/Relational Operator'
   *  RelationalOperator: '<Root>/Relational Operator1'
   *  Switch: '<Root>/ Switch2'
   */
  if (rtwdemo_condinput_U.input >= 5.0) {
    /* Outport: '<Root>/output' incorporates:
     *  Constant: '<Root>/  C10'
     *  Sum: '<Root>/ Sum1'
     */
    rtwdemo_condinput_Y.output = rtwdemo_condinput_U.input + 10.0;
  } else if (rtwdemo_condinput_U.input >= -5.0) {
    /* Outport: '<Root>/output' incorporates:
     *  Gain: '<Root>/  G3'
     *  Switch: '<Root>/ Switch2'
     */
    rtwdemo_condinput_Y.output = 3.0 * rtwdemo_condinput_U.input;
  } else {
    /* Outport: '<Root>/output' incorporates:
     *  Constant: '<Root>/ C_10'
     *  Sum: '<Root>/ Sum'
     *  Switch: '<Root>/ Switch2'
     */
    rtwdemo_condinput_Y.output = rtwdemo_condinput_U.input + -10.0;
  }

  /* End of Switch: '<Root>/Switch1' */
}
```

The generated code contains one `if` statement. The generated code for `Switch1:Path2` only executes if the `if` statement evaluates to false.

Close Model and Code Generation Report

```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

See Also

"Conditional input branch execution" (Simulink) | Multiport Switch | Switch

Related Examples

- "Optimization Tools and Techniques" on page 53-7

- “Eliminate Dead Code Paths in Generated Code” on page 53-61

Optimize Generated Code for Complex Signals

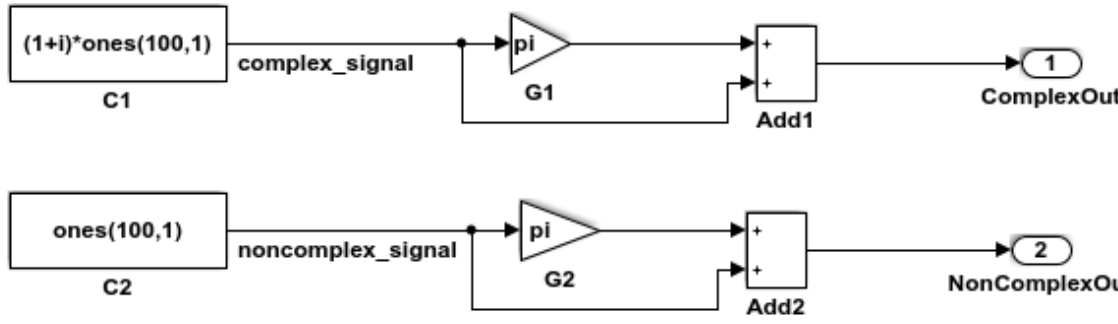
This example shows how Simulink Coder handles complex signals efficiently. To view the data types of the signals, update the model using **Simulation > Update Diagram**. Complex signals are represented as structures in generated code. Simulink Coder performs various optimizations on these structures. For example:

- Expression Folding: Gain and Sum operations on the complex signal are folded into a single expression.
- For-loop fusion: Two separate for-loops, one for the complex signal and one for noncomplex signal, are combined into a single for-loop.
- Inlined block parameters: The value of Gain block "pi" is inlined in the expression of the complex Gain-Sum.

Because of optimizations such as these, the code generated for complex and noncomplex signals is equally efficient.

Example Model

```
model='rtwdemo_complex';
open_system(model);
```



This model shows how Simulink Coder handles complex signals efficiently. To view the data types of the signals, update the model using Simulation > Update Diagram. Complex signals are represented as structures in generated code. Simulink Coder performs various optimizations on these structures. For example,

- o Expression folding: Gain and Sum operations on the complex signal are folded into a single expression.
- o For-loop fusion: Two separate for-loops, one for the complex signal and one for the noncomplex signal, are combined into a single for-loop.
- o Inlined block parameters: The value of Gain block "pi" is inlined in the expression of the complex Gain-Sum.

Because of optimizations such as these, the code generated for complex and noncomplex signals is equally efficient.

Generate Code Using Simulink Coder (double-click) **Generate Code Using Embedded Coder (double-click)**

Related Examples

- “Optimization Tools and Techniques” on page 53-7
- “Minimize Computations and Storage for Intermediate Results at Block Outputs” on page 53-36
- “Disable Nonfinite Checks or Inlining for Math Functions” on page 53-30

Speed Up Linear Algebra in Code Generated from a MATLAB Function Block

To improve the execution speed of code generated for certain linear algebra functions in a MATLAB Function block, specify that the code generator produce LAPACK calls. LAPACK is a software library for numerical linear algebra. The code generator uses the LAPACKE C interface to LAPACK. If you specify that you want to generate LAPACK calls, and the input arrays for the linear algebra functions meet certain criteria, the code generator produces the LAPACK calls. Otherwise, the code generator produces code for the linear algebra functions.

The code generator uses the LAPACK library that you specify. Specify a LAPACK library that is optimized for your execution environment. See www.netlib.org/lapack/faq.html#_what_and_where_are_the_lapack_vendors_implementations.

Specify LAPACK Library

To generate LAPACK calls, you must have access to a LAPACK callback class. A LAPACK callback class specifies the LAPACK library and LAPACKE header file for the LAPACK calls. To indicate that you want to generate LAPACK calls and that you want to use a specific LAPACK library, specify the name of the LAPACK callback class. In the Configuration Parameters dialog box, in the **Code Generation** category, on the **All Parameters** tab, set **Custom LAPACK library callback** to the name of the callback class, for example, `useMyLAPACK`.

Write LAPACK Callback Class

To specify the locations of a particular LAPACK library and LAPACKE header file, write a LAPACK callback class. Share the callback class with others who want to use this LAPACK library for LAPACK calls in generated code.

The callback class must derive from the abstract class coder.LAPACKCallback. Use the following example callback class as a template.

```
classdef useMyLAPACK < coder.LAPACKCallback
    methods (Static)
        function hn = getHeaderFilename()
            hn = 'mylapacke_custom.h';
        end
```

```
        function updateBuildInfo(buildInfo, buildctx)
            buildInfo.addIncludePaths(fullfile(pwd,'include'));
            libName = 'mylapack';
            libPath = fullfile(pwd,'lib');
            [~,linkLibExt] = buildctx.getStdLibInfo();
            buildInfo.addLinkObjects([libName linkLibExt], libPath, ...
                '', true, true);
            buildInfo.addDefines('HAVE_LAPACK_CONFIG_H');
            buildInfo.addDefines('LAPACK_COMPLEX_STRUCTURE');
        end
    end
end
```

You must provide the `getHeaderFilename` and `updateBuildInfo` methods. The `getHeaderFilename` method returns the LAPACKE header file name. In the example callback class, replace `mylapacke_custom.h` with the name of your LAPACKE header file. The `updateBuildInfo` method provides the information required for the build process to link to the LAPACK library. Use code like the code in the template to specify the location of header files and the full path name of the LAPACK library. In the example callback class, replace `mylapack` with the name of your LAPACK library.

If your compiler supports only complex data types that are represented as structures, include these lines in the `updateBuildInfo` method.

```
buildInfo.addDefines('HAVE_LAPACK_CONFIG_H');
buildInfo.addDefines('LAPACK_COMPLEX_STRUCTURE');
```

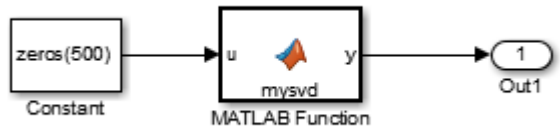
Generate LAPACK Calls by Specifying a LAPACK Callback Class

This example shows how to generate code that calls LAPACK functions in a specific LAPACK library. For this example, assume that the LAPACK callback class `useMyLAPACK` specifies the LAPACK library that you want.

1. Create a Simulink model.
2. Add a MATLAB Function block to the model.
3. In the MATLAB Function block, add code that calls a linear algebra function. For example, add the function `mysvd` that calls the MATLAB function `svd`.

   ```
   function s = mysvd(A)
       %#codegen
       s = svd(A);
   end
   ```

4 Add a Constant block to the left of the MATLAB Function block. Set the value to `zeros(500)`.
5 Add an Outport block to the right of the MATLAB Function block.
6 Connect the blocks.



7 In the Configuration Parameters dialog box, in the **Code Generation** category, on the **All Parameters** tab, set **Custom LAPACK library callback** to `useMyLAPACK`.

   The callback class must be on the MATLAB path.
8 Build the model.

If the input to `mysvd` is large enough, the code generator produces a LAPACK call for `svd`. Here is an example of a call to the LAPACK library function for `svd`.

```
info_t = LAPACKE_dgesvd(LAPACK_COL_MAJOR, 'N', 'N', (lapack_int)500,
    (lapack_int)500, &A[0], (lapack_int)500, &S[0], NULL, (lapack_int)1, NULL,
    (lapack_int)1, &superb[0]);
```

Locate LAPACK Library in Execution Environment

The LAPACK library must be available in your execution environment. If your LAPACK library is shared, use environment variables or linker options to specify the location of the LAPACK library.

- On a Windows platform, modify the PATH environment variable.
- On a Linux platform, modify the LD_LIBRARY_PATH environment variable or use the `rpath` linker option.
- On a Mac OS X, modify the DYLD_LIBRARY_PATH environment variable or use the `rpath` linker option.

To specify the `rpath` linker option, you can use the build information `addLinkFlags` method in the `updateBuildInfo` method of your coder.LAPACKCallback class. For example, for a GCC compiler:

```
buildInfo.addLinkFlags(sprintf('-Wl,-rpath,"%s"',libPath));
```

See Also

coder.LAPACKCallback

More About

- "LAPACK Calls for Linear Algebra in a MATLAB Function Block" (Simulink)

External Websites

- www.netlib.org/lapack
- www.netlib.org/lapack/faq.html#_what_and_where_are_the_lapack_vendors_implementations

Control Memory Allocation for Variable-Size Arrays in a MATLAB Function Block

Dynamic memory allocation allocates memory on the heap as needed at run time, instead of allocating memory statically on the stack. You can use dynamic memory allocation for arrays inside a MATLAB Function block.

You cannot use dynamic memory allocation for:

- Input and output signals. Variable-size input and output signals must have an upper bound.
- Parameters or global variables. Parameters and global variables must be fixed-size.
- Fields of bus arrays. Bus arrays cannot have variable-size fields.
- Discrete state properties of System objects associated with a MATLAB System block.

Dynamic memory allocation is beneficial when:

- You do not know the upper bound of an array.
- You do not want to allocate memory on the stack for large arrays.

Dynamic memory allocation and the freeing of this memory can result in slower execution of the generated code. To control the use of dynamic memory allocation for variable-size arrays in a MATLAB Function block, you can:

- Provide upper bounds for variable-size arrays.
- Disable dynamic memory allocation for MATLAB Function blocks.
- Modify the dynamic memory allocation threshold.

Provide Upper Bounds for Variable-Size Arrays

For an unbounded variable-size array, the code generator allocates memory dynamically on the heap. For a bounded variable-size array, if the size, in bytes, is less than the dynamic memory allocation threshold, the code generator allocates memory statically on the stack. To avoid dynamic memory allocation, provide upper bounds for the array dimensions so that the size of the array, in bytes, is less than the dynamic memory allocation threshold. See “Specify Upper Bounds for Variable-Size Arrays” (Simulink).

Disable Dynamic Memory Allocation for MATLAB Function Blocks

By default, dynamic memory allocation for MATLAB Function blocks is enabled for GRT-based targets and disabled for ERT-based targets. To change the setting, in the Configuration Parameters dialog box, on the **All Parameters** tab, in the **Simulation Target > Advanced parameters** category, clear or select the **Dynamic memory allocation in MATLAB Function blocks** check box.

If you disable dynamic memory allocation, you must provide upper bounds for variable-size arrays.

Modify the Dynamic Memory Allocation Threshold

Instead of disabling dynamic memory allocation for all variable-size arrays, you can use the dynamic memory allocation threshold to specify when the code generator uses dynamic memory allocation.

Use the dynamic memory allocation threshold to:

- Disable dynamic memory allocation for smaller arrays. For smaller arrays, static memory allocation can speed up generated code. However, static memory allocation can lead to unused storage space. You can decide that the unused storage space is not a significant consideration for smaller arrays.
- Enable dynamic memory allocation for larger arrays. For larger arrays, when you use dynamic memory allocation, you can significantly reduce storage requirements.

The default value of the dynamic memory allocation threshold is 64 kilobytes. To change the threshold, in the Configuration Parameters dialog box, on the **All Parameters** tab, in the **Simulation Target > Advanced parameters** category, set the **Dynamic memory allocation threshold in MATLAB Function blocks** parameter.

To use dynamic memory allocation for all variable-size arrays, set the threshold to `0`.

More About

- “Code Generation for Variable-Size Arrays” (Simulink)
- “Specify Upper Bounds for Variable-Size Arrays” (Simulink)
- “Use Dynamic Memory Allocation for Variable-Size Arrays in a MATLAB Function Block” (Simulink)

Optimize Memory Usage for Time Counters

This example shows how to optimize the amount of memory that the code generator allocates for time counters. The example optimizes the memory that stores elapsed time, the interval of time between two events.

The code generator represents time counters as unsigned integers. The word size of time counters is based on the setting of the model configuration parameter **Application lifespan (days)**, which specifies the expected maximum duration of time the application runs. You can use this parameter to prevent time counter overflows. The default size is 64 bits.

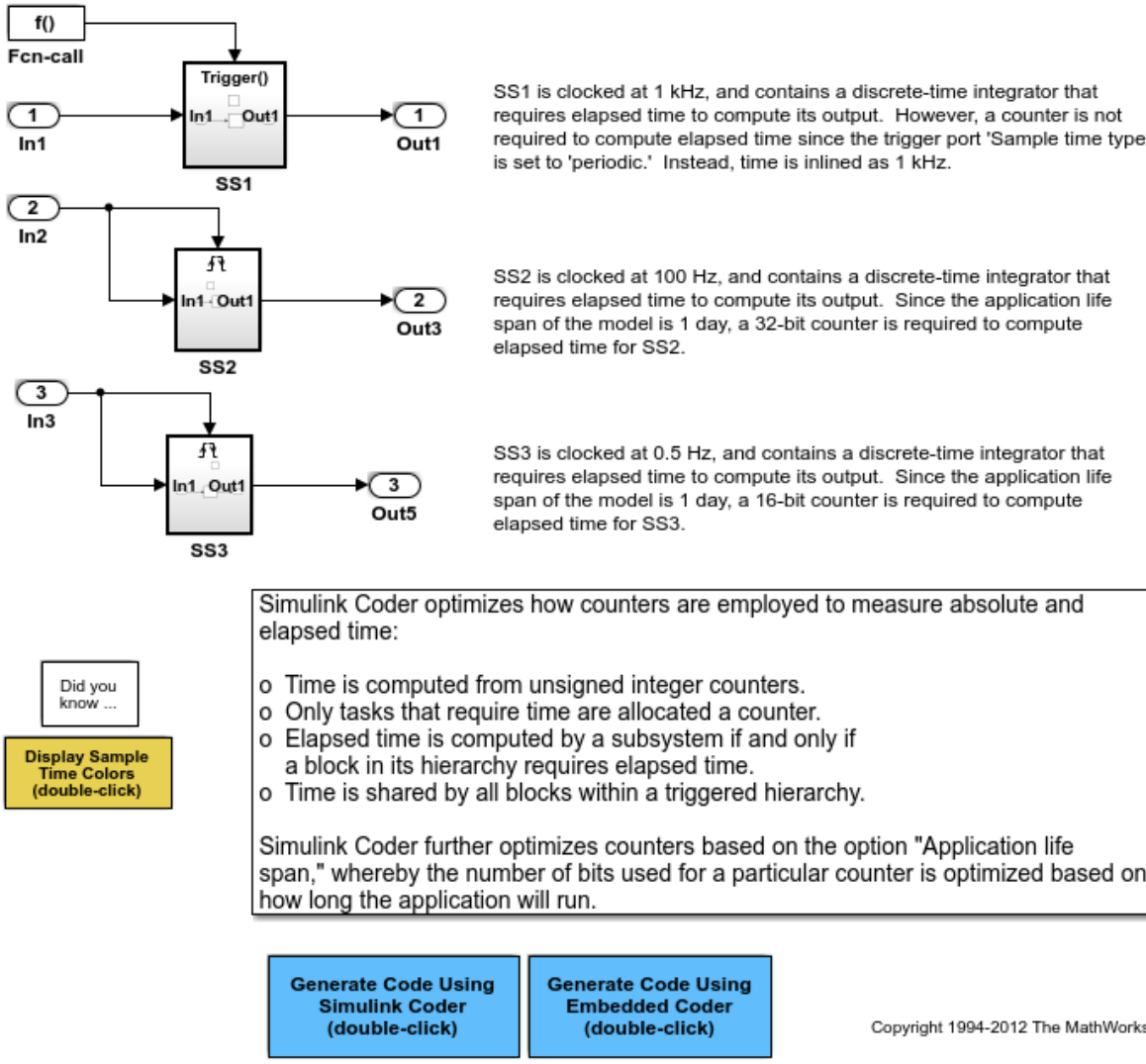
The number of bits that a time counter uses depends on the setting of the **Application lifespan (days)** parameter. For example, if a time counter increments at a rate of 1 kHz, to avoid an overflow, the counter has the following number of bits:

- Lifespan < 0.25 sec: 8 bits
- Lifespan < 1 min: 16 bits
- Lifespan < 49 days: 32 bits
- Lifespan > 50 days: 64 bits

A 64-bit time counter does not overflow for 590 million years.

Open Example Model

Open the example model `rtwdemo_abstime`.



The model consists of three subsystems `SS1`, `SS2`, and `SS3`. On the **Optimization** tab, the **Application lifespan (days)** parameter is set to the default, which is `auto`.

The three subsystems contain a discrete-time integrator that requires elapsed time as input to compute its output value. The subsystems vary as follows:

- SS1 - Clocked at 1 kHz. Does not require a time counter. **Sample time type** parameter for trigger port is set to `periodic`. Elapsed time is inlined as 0.001.
- SS2 - Clocked at 100 Hz. Requires a time counter. Based on a lifespan of 1 day, a 32-bit counter stores the elapsed time.
- SS3 - Clocked at 0.5 Hz. Requires a time counter. Based on a lifespan of 1 day, a 16-bit counter stores the elapsed time.

Simulate the Model

Simulate the model. By default, the model is configured to show sample times in different colors. Discrete sample times for the three subsystems appear red, green, and blue. Triggered subsystems are blue-green.

Generate Code and Report

1. Create a temporary folder for the build and inspection process.

2. Configure the model for the code generator to use the GRT system target file and a lifespan of `inf` days.

3. Build the model.

```
### Starting build procedure for model: rtwdemo_abstime
### Successful completion of build procedure for model: rtwdemo_abstime
```

Review Generated Code

Open the generated source file `rtwdemo_abstime.c`.

```
struct tag_RTM_rtwdemo_abstime_T {
  const char_T *errorStatus;

  /*
   * Timing:
   * The following substructure contains information regarding
   * the timing information for the model.
   */
  struct {
    uint32_T clockTick1;
    uint32_T clockTickH1;
    uint32_T clockTick2;
    uint32_T clockTickH2;
    struct {
```

```
      uint16_T TID[3];
      uint16_T cLimit[3];
    } TaskCounters;
  } Timing;
};
```

Four 32-bit unsigned integers, `clockTick1` , `clockTickH1` , `clockTick2` , and `clockTickH2` are counters for storing the elapsed time of subsystems `SS2` and `SS3`.

Enable Optimization and Regenerate Code

1. Reconfigure the model to set the lifespan to 1 day.

2. Build the model.

```
### Starting build procedure for model: rtwdemo_abstime
### Successful completion of build procedure for model: rtwdemo_abstime
```

Review the Regenerated Code

```
struct tag_RTM_rtwdemo_abstime_T {
  const char_T *errorStatus;

  /*
   * Timing:
   * The following substructure contains information regarding
   * the timing information for the model.
   */
  struct {
    uint32_T clockTick1;
    uint16_T clockTick2;
    struct {
      uint16_T TID[3];
      uint16_T cLimit[3];
    } TaskCounters;
  } Timing;
};
```

The new setting for the **Application lifespan (days)** parameter instructs the code generator to set aside less memory for the time counters. The regenerated code includes:

- 32-bit unsigned integer, `clockTick1`, for storing the elapsed time of the task for `SS2`

- 16-bit unsigned integer, `clockTick2`, for storing the elapsed time of the task for `SS3`

Related Information

- "Optimization Pane: General" (Simulink)
- "Timers in Asynchronous Tasks" (Simulink Coder)
- "Time-Based Scheduling and Code Generation" (Simulink Coder)

More About

- "Optimization Tools and Techniques" on page 53-7
- "Control Memory Allocation for Time Counters" on page 53-11
- "Access Timers Programmatically" (Simulink Coder)
- "Generate Code for an Elapsed Time Counter" (Simulink Coder)
- "Absolute Time Limitations" (Simulink Coder)

Minimize Memory Requirements During Code Generation

When the code generator produces code, it creates an partial representation of your model (called *`model`*`.rtw`), which the Target Language Compiler parses to transform block computations, parameters, signals, and constant data into a high-level language, (for example, C). Parameters and data are normally copied into the *`model`*`.rtw` file, whether they originate in the model itself or come from variables or objects in a workspace.

Models which have large amounts of parameter and constant data (such as lookup tables) can tax memory resources and slow down code generation because of the need to copy their data to *`model`*`.rtw`. You can improve code generation speed by limiting the size of data that is copied by using a `set_param` command, described below.

Data vectors such as those for parameters, lookup tables, and constant blocks whose sizes exceed a specified value are not copied into the *`model`*`.rtw` file. In place of the data vectors, the code generator places a special reference key in the *`model`*`.rtw` file that enables the Target Language Compiler to access the data directly from the Simulink software and format it directly into the generated code. This results in maintaining only one copy of large data vectors in memory.

You can specify the maximum number of elements that a parameter or other data source can have for the code generator to represent it literally in the *`model`*`.rtw` file. Whenever this threshold size is exceeded, the product writes a reference to the data to the *`model`*`.rtw` file, rather than its values. The default threshold value is 10 elements, which you can verify with

```
get_param(0, 'RTWDataReferencesMinSize')
```

To set the threshold to a different value, type the following `set_param` function in the MATLAB Command Window:

```
set_param(0, 'RTWDataReferencesMinSize', <size>)
```

Provide an integer value for `size` that specifies the number of data elements above which reference keys are to be used in place of actual data values.

Related Examples

- “Increase Code Generation Speed” on page 53-3

Optimize Generated Code Using Boolean Data for Logical Signals

Optimize generated code by storing logical signals as Boolean data. When you select the model configuration parameter **Implement logic signals as Boolean data (vs. double)**, blocks that generate logic signals output Boolean signals.

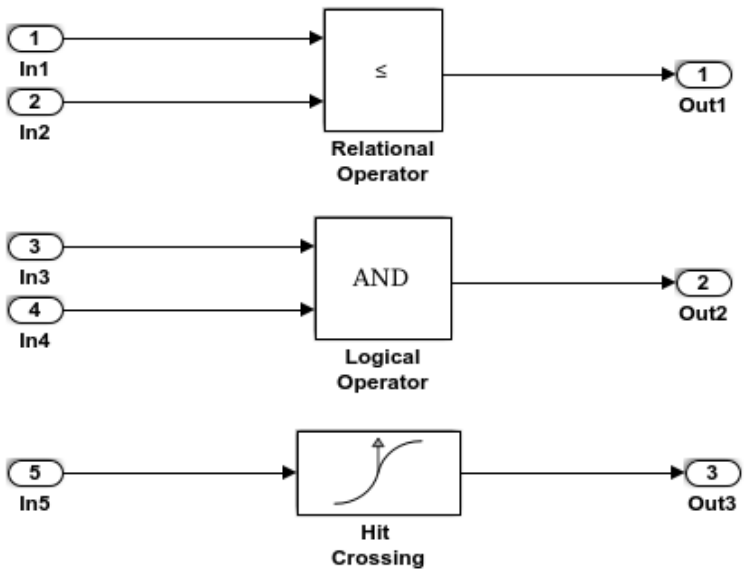
The optimization:

- Reduces the ROM and RAM consumption.
- Improves execution speed.

Example Model

Consider the model rtwdemo_logicalAsBoolean. The outputs of the `Relational Operator`, `Logical Operator` and `HitCrossing` blocks are `double`, even though they represent logical data.

```
model = 'rtwdemo_logicalAsBoolean';
open_system(model);
```



Generate Code

Create a temporary folder (in your system temporary folder) for the build and inspection process.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
```

Build the model.

```
rtwbuild(model)
```

```
### Starting build procedure for model: rtwdemo_logicalAsBoolean
### Successful completion of build procedure for model: rtwdemo_logicalAsBoolean
```

View the generated code without the optimization. These lines of code are in `rtwdemo_logicalAsBoolean.h`.

```
hfile = fullfile(cgDir,'rtwdemo_logicalAsBoolean_ert_rtw',...
    'rtwdemo_logicalAsBoolean.h');
rtwdemodbtype(hfile,'/* External outputs','/* Parameters (auto storage) */',1,0);
```

```
/* External outputs (root outports fed by signals with auto storage) */
typedef struct {
  real_T Out1;                         /* '<Root>/Out1' */
  real_T Out2;                         /* '<Root>/Out2' */
  real_T Out3;                         /* '<Root>/Out3' */
} ExtY_rtwdemo_logicalAsBoolean_T;
```

Enable Optimization

1. Open the Configuration Parameters dialog box.
2. On the **All Parameters** tab, select **Implement logic signals as Boolean data (vs. double)**.

Alternatively, you can use the command-line API to enable the optimization:

```
set_param(model,'BooleanDataType','on');
```

Generate Code with Optimization

The generated code stores the logical signal output as Boolean data.

Build the model.

```
rtwbuild(model)

### Starting build procedure for model: rtwdemo_logicalAsBoolean
### Successful completion of build procedure for model: rtwdemo_logicalAsBoolean
```

View the generated code with the optimization. These lines of code are in `rtwdemo_logicalAsBoolean.h`.

```
rtwdemodbtype(hfile,'/* External outputs','/* Parameters (auto storage) */',1,0);


/* External outputs (root outports fed by signals with auto storage) */
typedef struct {
  boolean_T Out1;                      /* '<Root>/Out1' */
  boolean_T Out2;                      /* '<Root>/Out2' */
  boolean_T Out3;                      /* '<Root>/Out3' */
} ExtY_rtwdemo_logicalAsBoolean_T;
```

Close the model and code generation report.

```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

See Also

"Implement logic signals as Boolean data (vs. double)" (Simulink)

Related Examples

- "Optimization Tools and Techniques" on page 53-7
- "Data Types Supported by Simulink" (Simulink)
- "Use Conditional Input Branch Execution" on page 53-67
- "Bitfields" on page 13-95

Reduce Memory Usage for Boolean and State Configuration Variables

1. Open the Model Configuration Parameters dialog box.
2. In the Model Configuration Parameters dialog box, select the **Optimization > Stateflow** pane.
3. Choose from these options:

   - **Use bitsets for storing state configuration** — Reduces the amount of memory that stores state configuration variables. However, it can increase the amount of memory that stores target code if the target processor does not include instructions for manipulating bitsets.
   - **Use bitsets for storing Boolean data** — Reduces the amount of memory that stores Boolean variables. However, it can increase the amount of memory that stores target code if the target processor does not include instructions for manipulating bitsets.

   Note: You cannot use bitsets when you generate code for these cases:

   - An external mode simulation
   - A target that specifies an explicit structure alignment

See Also

"Use bitsets for storing state configuration" (Simulink) | "Use bitsets for storing Boolean data" (Simulink)

Related Examples

- "Optimization Tools and Techniques" on page 53-7
- "Optimize Generated Code Using Boolean Data for Logical Signals" on page 53-87
- "Bitfields" on page 13-95

Customize Stack Space Allocation

Your application might be constrained by limited memory. Controlling the maximum allowable size for the stack is one way to modify whether data is defined as local or global in the generated code. You can limit the use of stack space by specifying a positive integer value for the "Maximum stack size (bytes)" (Simulink) parameter, on the **Optimization > Signals and Parameters** pane of the Configuration parameter dialog box. Specifying the maximum allowable stack size provides control over the number of local and global variables in the generated code. Specifically, lowering the maximum stack size might generate more variables into global structures. The number of local and global variables help determine the required amount of stack space for execution of the generated code.

The default setting for "Maximum stack size (bytes)" (Simulink) is `Inherit from target`. In this case, the value of the maximum stack size is the smaller value of the following: the default value set by the code generator (200,000 bytes) or the value of the TLC variable `MaxStackSize` found in the system target file (`ert.tlc`).

To specify a smaller stack size for your application, select the `Specify a value` option of the **Maximum stack size (bytes)** parameter and enter a positive integer value. To specify a smaller stack size at the command line, use:

```
set_param(model_name, 'MaxStackSize', 65000);
```

Note: For overall executable stack usage metrics, you might want to do a target-specific measurement, such as using runtime (empirical) analysis or static (code path) analysis with object code.

It is recommended that you use the **Maximum stack size (bytes)** parameter to control stack space allocation instead of modifying the TLC variable, `MaxStackSize`, in the system target file. However, a target author might want to set the TLC variable, `MaxStackSize`, for a target. To set `MaxStackSize`, use `assign` statements in the system target file (`ert.tlc`), as in the following example.

```
%assign MaxStackSize = 4096
```

Write your `%assign` statements in the `Configure RTW code generation settings` section of the system target file. The `%assign` statement is described in "Target Language Compiler" (Simulink Coder).

See Also

"Maximum stack size (bytes)" (Simulink)

Related Examples

- "Optimization Tools and Techniques" on page 53-7
- "Enable and Reuse Local Block Outputs in Generated Code" on page 53-100
- "Minimize Computations and Storage for Intermediate Results at Block Outputs" on page 53-36
- "Inline Numeric Values of Block Parameters" on page 53-43

Optimize Generated Code Using `memset` Function

This example shows how to optimize the generated code by using the `memset` function to clear the internal storage. When you select the model configuration parameter **Use memset to initialize floats and doubles to 0.0**, the `memset` function clears internal storage, regardless of type, to the integer bit pattern `0` (that is, all bits are off).

If your compiler and target CPU both represent floating-point zero with the integer bit pattern `0`, consider setting this parameter to gain execution and ROM efficiency.

NOTE: The command-line values are the reverse of the settings values. 'on' in the command line corresponds to clearing the setting. `'off'` in the command line corresponds to selecting the setting.

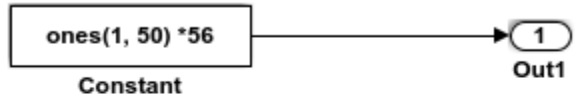
This optimization:

- Reduces ROM consumption.
- Improves execution speed.

Example Model

Consider the model matlab:rtwdemo_memset.

```
model = 'rtwdemo_memset';
open_system(model);
```



Generate Code

The code generator uses a loop to initialize the `Constant` block values.

Create a temporary folder (in your system temporary folder) for the build and inspection process.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
```

Build the model.

```
rtwbuild(model)
```

```
### Starting build procedure for model: rtwdemo_memset
### Successful completion of build procedure for model: rtwdemo_memset
```

View the generated code without the optimization. These lines of code are in `rtwdemo_memset.c`.

```
cfile = fullfile(cgDir,'rtwdemo_memset_grt_rtw','rtwdemo_memset.c');
rtwdemodbtype(cfile,'/* Model initialize function */',...
    '/* Model terminate function */',1,0);
```

```
/* Model initialize function */
void rtwdemo_memset_initialize(void)
{
  /* Registration code */

  /* initialize error status */
  rtmSetErrorStatus(rtwdemo_memset_M, (NULL));

  /* external outputs */
  {
    int32_T i;
    for (i = 0; i < 50; i++) {
      rtwdemo_memset_Y.Out1[i] = 0.0;
    }
  }

  {
    int32_T i;

    /* ConstCode for Outport: '<Root>/Out1' */
    for (i = 0; i < 50; i++) {
      rtwdemo_memset_Y.Out1[i] = 56.0;
    }

    /* End of ConstCode for Outport: '<Root>/Out1' */
  }
}
```

Enable Optimization

1. Open the Configuration Parameters dialog box.
2. On the **All Parameters** tab, select **Use memset to initialize floats and doubles to 0.0**.

Alternatively, you can use the command-line API to enable the optimization:

```
set_param(model,'InitFltsAndDblsToZero','off');
```

Generate Code with Optimization

The code generator uses the `memset` function to initialize the `Constant` block values.

Build the model.

```
rtwbuild(model)
```

```
### Starting build procedure for model: rtwdemo_memset
### Successful completion of build procedure for model: rtwdemo_memset
```

View the generated code with the optimization. These lines of code are in `rtwdemo_memset.c`.

```
rtwdemodbtype(cfile,'/* Model initialize function */',...
    '/* Model terminate function */',1,0);
```

```
/* Model initialize function */
void rtwdemo_memset_initialize(void)
{
  /* Registration code */

  /* initialize error status */
  rtmSetErrorStatus(rtwdemo_memset_M, (NULL));

  /* external outputs */
  (void) memset(&rtwdemo_memset_Y.Out1[0], 0,
                50U*sizeof(real_T));

  {
    int32_T i;

    /* ConstCode for Outport: '<Root>/Out1' */
    for (i = 0; i < 50; i++) {
```

```
      rtwdemo_memset_Y.Out1[i] = 56.0;
    }

    /* End of ConstCode for Outport: '<Root>/Out1' */
  }
}
```

Close the model and the code generation report.

```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

See Also

"Use memset to initialize floats and doubles to 0.0" (Simulink)

Related Examples

- "Optimization Tools and Techniques" on page 53-7
- "Use `memcpy` Function to Optimize Generated Code for Vector Assignments" on page 53-52
- "Vector Operation Optimization" on page 53-97
- "Remove Initialization Code" on page 56-3

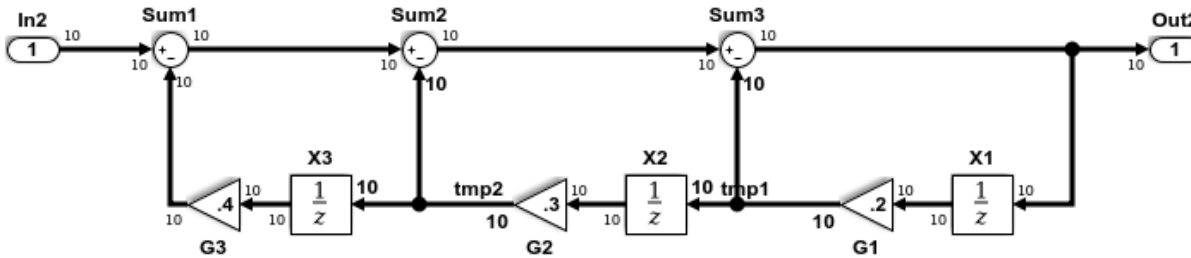
Vector Operation Optimization

This example shows how Simulink® Coder™ optimizes generated code by setting block output that generates vectors to scalars, for blocks such as the Mux, Sum, Gain, and Bus. This optimization reduces stack memory by replacing temporary local arrays with local variables.

Example Model

In the model, rtwdemo_VectorOptimization, the output of Gain blocks `G1` and `G2` are the vector signals `tmp1` and `tmp2`. These vectors have a width of 10.

```
model = 'rtwdemo_VectorOptimization';
open_system(model);
set_param(model, 'SimulationCommand', 'update')
```



Generate Code

Create a temporary folder (in your system temporary folder) for the build and inspection process.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
```

Build the model.

```
rtwbuild(model)
```

```
### Starting build procedure for model: rtwdemo_VectorOptimization
### Successful completion of build procedure for model: rtwdemo_VectorOptimization
```

The optimized code is in `rtwdemo_VectorOptimization.c`. The signals `tmp1` and `tmp2` are the local variables `rtb_tmp1` and `rtb_tmp2`.

```
cfile = fullfile(cgDir,'rtwdemo_VectorOptimization_grt_rtw',...
    'rtwdemo_VectorOptimization.c');
rtwdemodbtype(cfile,'/* Model step', '/* Model initialize', 1, 0);
```

```
/* Model step function */
void rtwdemo_VectorOptimization_step(void)
{
  int32_T i;
  real_T rtb_tmp2;
  real_T rtb_tmp1;
  real_T rtb_Sum3;
  for (i = 0; i < 10; i++) {
    /* Gain: '<Root>/G2' incorporates:
     *  UnitDelay: '<Root>/X2'
     */
    rtb_tmp2 = 0.3 * rtwdemo_VectorOptimization_DW.X2_DSTATE[i];

    /* Gain: '<Root>/G1' incorporates:
     *  UnitDelay: '<Root>/X1'
     */
    rtb_tmp1 = 0.2 * rtwdemo_VectorOptimization_DW.X1_DSTATE[i];

    /* Sum: '<Root>/Sum3' incorporates:
     *  Gain: '<Root>/G3'
     *  Inport: '<Root>/In2'
     *  Sum: '<Root>/Sum1'
     *  Sum: '<Root>/Sum2'
     *  UnitDelay: '<Root>/X3'
     */
    rtb_Sum3 = ((rtwdemo_VectorOptimization_U.In2[i] - 0.4 *
                 rtwdemo_VectorOptimization_DW.X3_DSTATE[i]) - rtb_tmp2) -
      rtb_tmp1;

    /* Outport: '<Root>/Out2' */
    rtwdemo_VectorOptimization_Y.Out2[i] = rtb_Sum3;

    /* Update for UnitDelay: '<Root>/X3' */
    rtwdemo_VectorOptimization_DW.X3_DSTATE[i] = rtb_tmp2;
```

```
    /* Update for UnitDelay: '<Root>/X2' */
    rtwdemo_VectorOptimization_DW.X2_DSTATE[i] = rtb_tmp1;

    /* Update for UnitDelay: '<Root>/X1' */
    rtwdemo_VectorOptimization_DW.X1_DSTATE[i] = rtb_Sum3;
  }
}
```

Close the model and code generation report.

```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

Related Examples

- “Optimization Tools and Techniques” on page 53-7
- “Minimize Computations and Storage for Intermediate Results at Block Outputs” on page 53-36
- “Use `memcpy` Function to Optimize Generated Code for Vector Assignments” on page 53-52

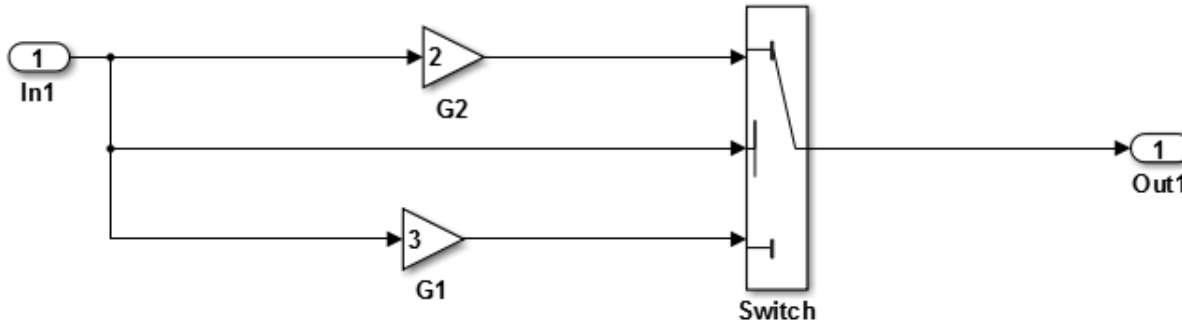
Enable and Reuse Local Block Outputs in Generated Code

In this section...

This example shows how to specify block output as local variables. The code generator can potentially reuse these local variables in the generated code. Declaring block output as local variables conserves ROM consumption. Reusing local variables conserves RAM consumption, reduces data copies, and increases execution speed.

Example Model

1 Use Inport, Outport, Gain, and Switch blocks to create the following model. In this example, the model is named `local_variable_ex`.



2 For `G2`, open the Gain Block Parameters dialog box. Enter a value of `2`.

3 For `G1`, enter a value of `3`.

4 For the Switch block, open the Block Parameters dialog box. For the **Criteria for passing first input** parameter, select `u2>=Threshold`.

Generate Code Without Optimization

1. Open the Model Configuration Parameters dialog box. Select the **Solver** pane. For the **Type** parameter, select `Fixed-step`.
2. Select the **All Parameters** tab and clear **Signal Storage Reuse**.
3. Select the **Code Generation > Report** pane and select **Create code generation report**.
4. Select the **Code Generation** pane. Select **Generate code only**, and then, in the model window, press **Ctrl+B**. When code generation is complete, an HTML code generation report appears.
5. In the code generation report, select the `local_variable_ex.c` section and view the model step function. The Gain block outputs are the global variables `local_variable_ex_B.G2` and `local_variable_ex_B.G1`.

```
/* Model step function */
void local_variable_ex_step(void)
{
  /* Switch: '<Root>/Switch' incorporates:
   *  Inport: '<Root>/In1'
   */
  if (local_variable_ex_U.In1 >= 0.0) {
    /* Gain: '<Root>/G2' */
    local_variable_ex_B.G2 = 2.0 * local_variable_ex_U.In1;

    /* Outport: '<Root>/Out1' */
    local_variable_ex_Y.Out1 = local_variable_ex_B.G2;
  } else {
    /* Gain: '<Root>/G1' */
    local_variable_ex_B.G1 = 3.0 * local_variable_ex_U.In1;

    /* Outport: '<Root>/Out1' */
    local_variable_ex_Y.Out1 = local_variable_ex_B.G1;
  }

  /* End of Switch: '<Root>/Switch' */
```

Enable Local Block Outputs and Generate Code

1. Open the Model Configuration Parameters dialog box. On the **All Parameters** tab, select **Signal Storage Reuse**. **Signal Storage Reuse** enables the following optimization parameters:

- **Enable local block outputs**
- **Reuse local block outputs**
- **Eliminate superfluous local variables (expression folding)**

2 Clear **Reuse local block outputs** and **Eliminate superfluous local variables (expression folding)**.

3 Generate code and view the model step function. There are three local variables in the model step function because you selected the optimization parameter **Enable Local Block Outputs**. The local variables `rtb_G2` and `rtb_G1` hold the outputs of the Gain blocks. The local variable `rtb_Switch` holds the output of the Switch block.

```
/* Model step function */
void local_variable_ex_step(void)
{
  real_T rtb_Switch;
  real_T rtb_G2;
  real_T rtb_G1;

  /* Switch: '<Root>/Switch' incorporates:
   *  Inport: '<Root>/In1'
   */
  if (local_variable_ex_U.In1 >= 0.0) {
    /* Gain: '<Root>/G2' */
    rtb_G2 = 2.0 * local_variable_ex_U.In1;
    rtb_Switch = rtb_G2;
  } else {
    /* Gain: '<Root>/G1' */
    rtb_G1 = 3.0 * local_variable_ex_U.In1;
    rtb_Switch = rtb_G1;
  }

  /* End of Switch: '<Root>/Switch' */

  /* Outport: '<Root>/Out1' */
  local_variable_ex_Y.Out1 = rtb_Switch;
```

Reuse Local Block Outputs and Generate Code

1 Open the Model Configuration Parameters dialog box. On the **All Parameters** tab, select **Reuse local block outputs**.

2 Generate code. In the `local_variable_ex.c` section, view the model step function. There is one local variable, `rtb_G2`, that the code generator uses three times.

```
/* Model step function */
void local_variable_ex_step(void)
{
  real_T rtb_G2;

  /* Switch: '<Root>/Switch' incorporates:
   *  Inport: '<Root>/In1'
   */
  if (local_variable_ex_U.In1 >= 0.0) {
    /* Gain: '<Root>/G2' */
    rtb_G2 = 2.0 * local_variable_ex_U.In1;
  } else {
    /* Gain: '<Root>/G1' */
    rtb_G2 = 3.0 * local_variable_ex_U.In1;
  }

  /* End of Switch: '<Root>/Switch' */

  /* Outport: '<Root>/Out1' */
  local_variable_ex_Y.Out1 = rtb_G2;
```

The extra temporary variable `rtb_Switch` and the associated data copy is not in the generated code.

See Also

"Enable local block outputs" (Simulink)

Related Examples

- "Optimization Tools and Techniques" on page 53-7
- "Customize Stack Space Allocation" on page 53-91
- "Signal Representation in Generated Code" (Simulink Coder)

Configuration in Embedded Coder

Specify Global Variable Localization

When you generate code for a model, the code generator can optimize variable references by replacing global variables with local variables. Replacing global variables with local variables improves execution speed and reduces RAM/ROM. Creating more local variables can increase stack usage.

Some of the global variables that the code generator can localize include:

- Global signals that cross subsystem boundaries.
- Global signals across Simulink and Stateflow domains.
- Unused global state variables.
- Redundant local Data Store Memory block signals.

To enable the global variable localization analysis:

1. In the Configuration Parameters dialog box, on the **Code Generation** pane, in the **System target file** box, specify an ERT target.
2. Verify that the `OptimizeBlockIOStorage` parameter is set to '`on`':

   ```
   >> get_param(gcs,'OptimizeBlockIOStorage')
   ans =
         on
   ```

3. Verify that `AdvancedOptControl` is not set to '`-SLCI`':

   ```
   >> get_param(gcs,'AdvancedOptControl')
   ans =
         ''
   ```

4. Set the storage class for signals to `Auto`.

The code generator does not localize global variables for MATLAB system objects or AUTOSAR.

See Also

"Enable local block outputs" (Simulink)

Related Examples

- "Enable and Reuse Local Block Outputs in Generated Code" on page 53-100

- "Minimize Computations and Storage for Intermediate Results at Block Outputs" on page 53-36

Set Hardware Implementation Parameters

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

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

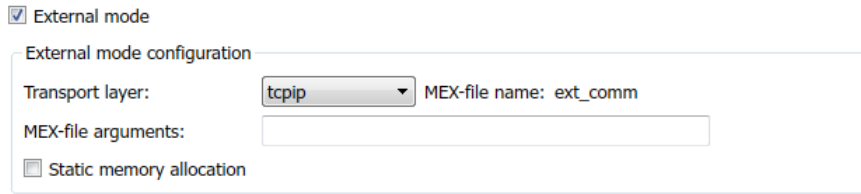
Before generating and deploying code, get familiar with the **Hardware Implementation** pane of the Configuration Parameters dialog box. By default, target hardware microprocessor device details are hidden. To view the details, click the **Device details** arrow. See “Hardware Implementation Pane” (Simulink) in the Simulink documentation and “Configure Run-Time Environment Options” (Simulink Coder) in the Simulink Coder documentation for more information.

You can use the example “Configure Target Hardware Characteristics” (Simulink Coder) to determine characteristics of your C or C++ compiler and target hardware. By using the example model with your target development system and debugger, you can observe the behavior of the code as it executes on the target hardware. You can then use the information to refine hardware target device parameters for your model.

Use External Mode with the ERT Target

Selecting the **External mode** option turns on generation of code to support external mode communication between host (Simulink) and target systems. The Embedded Coder software supports Simulink external mode simulation, as described in the Simulink Coder topic "Set Up and Use Host/Target Communication Channel" (Simulink Coder).

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



Memory Management

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

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

Generation of Pure Integer Code with External Mode

The Embedded Coder software supports generation of pure integer code when external mode code is generated. To do this, select the **External mode** option and deselect the **Support: floating-point numbers** option on the **Code Generation** > **Interface** pane.

This selection lets you generate external mode code that is free of 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 "Run the External Program" (Simulink Coder) in the Simulink Coder 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
  ```

Related Examples

- "Optimization Tools and Techniques" on page 53-7
- "Set Up and Use Host/Target Communication Channel" on page 41-2

Data Copy Reduction in Embedded Coder

Optimize Global Variable Usage

In this section...

To tune your application and choose tradeoffs for execution speed and memory usage, you can choose a global variable reference optimization for the generated code.

In the Configuration Parameters dialog box, on the **All Parameters** tab, in the **Optimize global data access** drop-down list, three parameter options control global variable usage optimizations.

- `None`. Use default optimizations. This choice works well for most models. The code generator balances the use of local and global variables. It generates code which balances RAM and ROM consumption and execution speed.
- `Use global to hold temporary results`. Reusing global variables improves code efficiency and readability. This optimization reuses global variables, which results in the code generator defining fewer variables. It reduces RAM and ROM consumption and data copies.
- `Minimize global data access`. Using local variables to cache global data reduces ROM consumption by reducing code size in certain cases, such as when the global variables are scalars. This optimization improves execution speed because the code uses fewer instructions for local variable references than for global variable references.

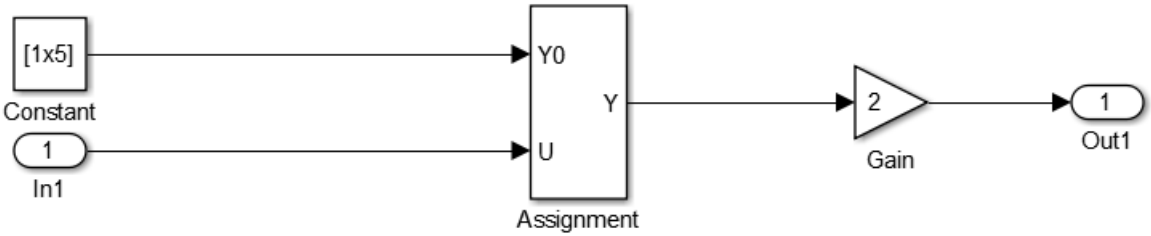
  Minimizing the use of global variables by using local variables interacts with stack usage control. For example, stack size can determine the number of local and global variables that the code generator can allocate in the generated code. For more information, see "Customize Stack Space Allocation" (Simulink Coder).

Use Global to Hold Temporary Results

The code generator uses global and local variables when you select `None` versus when you select `Use global to hold temporary results`.

Example Model

In the model matlab:rtwdemo_optimize_global_ebf, an Assignment block assigns values coming from the Inport and Constant blocks to an output signal. The output signal feeds into a Gain block.




```
model = 'rtwdemo_optimize_global_ebf';
load_system('rtwdemo_optimize_global_ebf')
```

Generate Code without Optimization

1. In the Configuration Parameters dialog box, on the All Parameters tab, verify that the **Signal storage reuse** parameter is selected.
2. In the Configuration Parameters dialog box, for the **Optimize global access parameter**, select `None` or enter the following command in the MATLAB Command Window:

```
set_param('rtwdemo_optimize_global_ebf','GlobalVariableUsage','None');
```

In your system's temporary folder, create a folder for the build and inspection process:

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
```

Build the model.

```
rtwbuild(model);
```

```
### Starting build procedure for model: rtwdemo_optimize_global_ebf
### Successful completion of build procedure for model: rtwdemo_optimize_global_ebf
```

View the generated code without the optimization. Here is a portion of `rtwdemo_optimize_global_ebf.c`.

```
cfile = fullfile(cgDir,'rtwdemo_optimize_global_ebf_ert_rtw',...
    'rtwdemo_optimize_global_ebf.c');
rtwdemodbtype(cfile,'/* Model step','/* Model initialize',1, 0);
```

```
/* Model step function */
void rtwdemo_optimize_global_ebf_step(void)
{
  real_T rtb_Assignment[5];
  int32_T i;

  /* Assignment: '<Root>/Assignment' incorporates:
   *  Constant: '<Root>/Constant'
   *  Inport: '<Root>/In1'
   */
  for (i = 0; i < 5; i++) {
    rtb_Assignment[i] = rtCP_Constant_Value[i];
  }

  rtb_Assignment[1] = rtU.In1;

  /* End of Assignment: '<Root>/Assignment' */

  /* Outport: '<Root>/Out1' incorporates:
   *  Gain: '<Root>/Gain'
   */
  for (i = 0; i < 5; i++) {
    rtY.Out1[i] = 2.0 * rtb_Assignment[i];
  }

  /* End of Outport: '<Root>/Out1' */
}
```

The code assigns values to the local vector `rtb_Assignment`. The last statement copies the values in the local vector `rtb_Assignment` to the global vector `rtY.Out1`. Fewer global variable references result in improved execution speed. The code uses more instructions for global variable references than for local variable references.

In the Static Code Metrics Report, examine the Global Variables section.

1. In the Code Generation Report window, select **Static Code Metrics Report**.
2. Scroll down to the `Global Variables` section.
3. Select the **[+]** sign before each variable to expand it.

Global Variable	Size (bytes)	Reads / Writes	Reads / Writes in a Function
[-] rtY	40	1	1
Out1	40	1	1
[-] rtU	8	1	1
In1	8	1	1
[-] rtM_	4	0*	0*
errorStatus	4	0	0
Total	**52**	**2**	

* The global variable is not directly used in any function.

The total number of reads and writes for global variables is 2.

Generate Code with Optimization

In the Configuration Parameters dialog box, for the **Optimize global access parameter**, select `Use global to hold temporary results`, or enter the following command in the MATLAB Command Window:

```
set_param('rtwdemo_optimize_global_ebf',...
    'GlobalVariableUsage','Use global to hold temporary results');
```

Build the model.

```
rtwbuild(model);
```

```
### Starting build procedure for model: rtwdemo_optimize_global_ebf
### Successful completion of build procedure for model: rtwdemo_optimize_global_ebf
```

View the generated code with the optimization. Here is a portion of `rtwdemo_optimize_global_ebf.c`.

```
cfile = fullfile(cgDir,'rtwdemo_optimize_global_ebf_ert_rtw',...
    'rtwdemo_optimize_global_ebf.c');
rtwdemodbtype(cfile,'/* Model step','/* Model initialize',1, 0);
```

```
/* Model step function */
void rtwdemo_optimize_global_ebf_step(void)
{
```

```
  int32_T i;

  /* Assignment: '<Root>/Assignment' incorporates:
   *  Constant: '<Root>/Constant'
   *  Inport: '<Root>/In1'
   */
  for (i = 0; i < 5; i++) {
    rtY.Out1[i] = rtCP_Constant_Value[i];
  }

  rtY.Out1[1] = rtU.In1;

  /* End of Assignment: '<Root>/Assignment' */

  /* Outport: '<Root>/Out1' incorporates:
   *  Gain: '<Root>/Gain'
   */
  for (i = 0; i < 5; i++) {
    rtY.Out1[i] *= 2.0;
  }

  /* End of Outport: '<Root>/Out1' */
}
```

The code assigns values to the global vector `rtY.Out1` without using a local variable. This assignment improves ROM and RAM consumption and reduces data copies. The code places the value in the destination variable for each assignment instead of copying the value at the end. In the Static Code Metrics Report, examine the Global Variables section.

Global Variable	Size (bytes)	Reads / Writes	Reads / Writes in a Function
[-] rtY	40	4	4
Out1	40	4	4
[-] rtU	8	1	1
In1	8	1	1
[-] rtM_	4	0*	0*
errorStatus	4	0	0
Total	**52**	**5**	

* The global variable is not directly used in any function.

As a result of using global variables to hold local results, the total number of reads and writes for global variables has increased from 2 to 5. This optimization reduces data copies by reusing global variables.

Close the code generation report.

```
rtwdemoclean;
cd(currentDir)
```

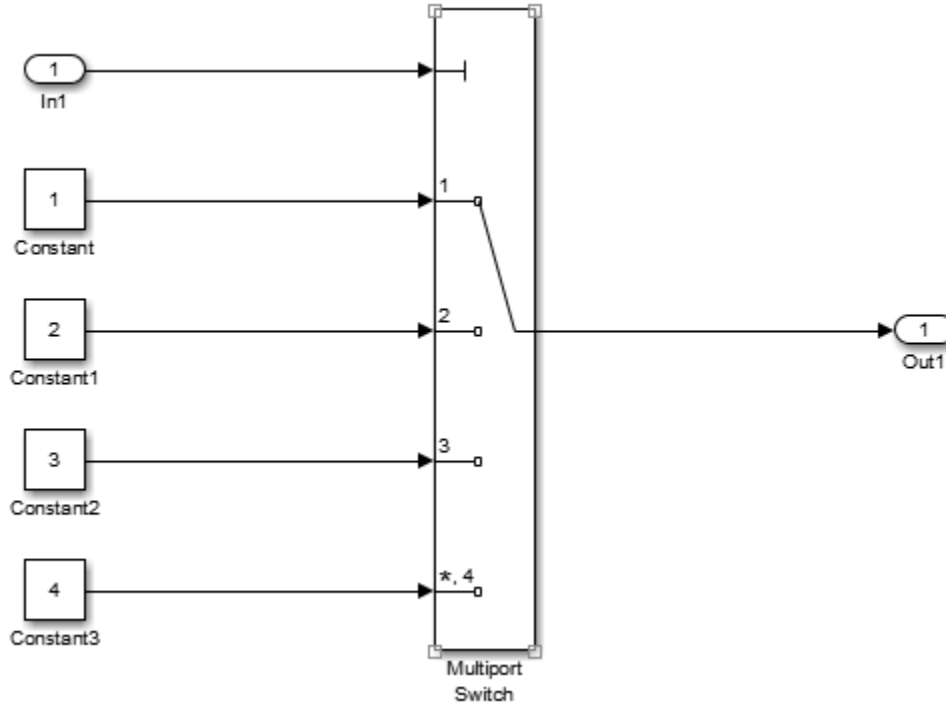
Minimize Global Data Access

Generate optimized code that reads from and writes to global variables less frequently.

Example Model

In the model matlab:rtwdemo_optimize_global, five signals feed into a Multiport Switch block.

```
model = 'rtwdemo_optimize_global';
load_system('rtwdemo_optimize_global')
```



Generate Code without Optimization

1. In the Configuration Parameters dialog box, on the **All Parameters** tab, verify that the **Signal storage reuse** parameter is selected.
2. In the Configuration Parameters dialog box, on the **All Parameters** tab, for the **Optimize global access** parameter, select `None` or enter the following command in the MATLAB Command Window:

```
set_param('rtwdemo_optimize_global','GlobalVariableUsage','None');
```

In your system's temporary folder, create a folder for the build and inspection process.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
```

Build the model.

```
rtwbuild(model);

### Starting build procedure for model: rtwdemo_optimize_global
### Successful completion of build procedure for model: rtwdemo_optimize_global
```

View the generated code without the optimization. Here is a portion of `rtwdemo_optimize_global.c`.

```
cfile = fullfile(cgDir,'rtwdemo_optimize_global_ert_rtw',...
    'rtwdemo_optimize_global.c');
rtwdemodbtype(cfile,'/* Model step','/* Model initialize',1, 0);


/* Model step function */
void rtwdemo_optimize_global_step(void)
{
  /* MultiPortSwitch: '<Root>/Multiport Switch' incorporates:
   *  Inport: '<Root>/In1'
   */
  switch ((int32_T)rtU.In1) {
   case 1:
    /* Outport: '<Root>/Out1' incorporates:
     *  Constant: '<Root>/Constant'
     */
    rtY.Out1 = 1.0;
    break;

   case 2:
    /* Outport: '<Root>/Out1' incorporates:
     *  Constant: '<Root>/Constant1'
     */
    rtY.Out1 = 2.0;
    break;

   case 3:
    /* Outport: '<Root>/Out1' incorporates:
     *  Constant: '<Root>/Constant2'
     */
    rtY.Out1 = 3.0;
    break;

   default:
    /* Outport: '<Root>/Out1' incorporates:
     *  Constant: '<Root>/Constant3'
     */
```

```
    rtY.Out1 = 4.0;
    break;
  }

  /* End of MultiPortSwitch: '<Root>/Multiport Switch' */
}
```

In the Static Code Metrics Report, examine the `Global Variables` section.

1. In the Code Generation Report window, select **Static Code Metrics Report**.
2. Scroll down to the `Global Variables` section.
3. Select the **[+]** sign before each variable to expand it.

Global variables defined in the generated code.

Global Variable	Size (bytes)	Reads / Writes	Reads / Writes in a Function
[−] rtU	8	1	1
In1	8	1	1
[−] rtY	8	4	4
Out1	8	4	4
[−] rtM	4	0*	0*
errorStatus	4	0	0
Total	**20**	**5**	

* The global variable is not directly used in any function.

The total number of reads and writes for global variables is 5.

Enable Optimization and Generate Code

On the **All Parameters** tab, for the **Optimize global data access** parameter, select `Minimize global data access` or enter the following command in the MATLAB Command Window:

```
set_param('rtwdemo_optimize_global',...
    'GlobalVariableUsage','Minimize global data access');
```

Build the model.

```
rtwbuild(model);

### Starting build procedure for model: rtwdemo_optimize_global
### Successful completion of build procedure for model: rtwdemo_optimize_global
```

View the generated code with the optimization. Here is a portion of `rtwdemo_optimize_global.c`.

```
cfile = fullfile(cgDir,'rtwdemo_optimize_global_ert_rtw',...
    'rtwdemo_optimize_global.c');
rtwdemodbtype(cfile,'/* Model step','/* Model initialize',1, 0);


/* Model step function */
void rtwdemo_optimize_global_step(void)
{
  real_T tmp_Out1;

  /* MultiPortSwitch: '<Root>/Multiport Switch' incorporates:
   *  Inport: '<Root>/In1'
   */
  switch ((int32_T)rtU.In1) {
   case 1:
    /* Outport: '<Root>/Out1' incorporates:
     *  Constant: '<Root>/Constant'
     */
    tmp_Out1 = 1.0;
    break;

   case 2:
    /* Outport: '<Root>/Out1' incorporates:
     *  Constant: '<Root>/Constant1'
     */
    tmp_Out1 = 2.0;
    break;

   case 3:
    /* Outport: '<Root>/Out1' incorporates:
     *  Constant: '<Root>/Constant2'
     */
    tmp_Out1 = 3.0;
    break;

   default:
    /* Outport: '<Root>/Out1' incorporates:
```

```
     *  Constant: '<Root>/Constant3'
     */
    tmp_Out1 = 4.0;
    break;
  }

  /* End of MultiPortSwitch: '<Root>/Multiport Switch' */

  /* Outport: '<Root>/Out1' */
  rtY.Out1 = tmp_Out1;
}
```

In `rtwdemo_optimize_global.c`, the code assigns a constant value to the local variable `tmp_Out1` in each case statement. The last statement in the code copies the value of `tmp_Out1` to the global variable `rtY.Out1`. Fewer global variable references result in fewer instructions and improved execution speed.

In the **Static Code Metrics Report**, examine the `Global Variables` section. As a result of minimizing global data accesses, the total number of reads and writes for global variables has decreased from 5 to 2.

Global variables defined in the generated code.

Global Variable	Size (bytes)	Reads / Writes	Reads / Writes in a Function
[−] rtU	8	1	1
In1	8	1	1
[−] rtY	8	1	1
Out1	8	1	1
[−] rtM	4	0*	0*
errorStatus	4	0	0
Total	**20**	**2**	

* The global variable is not directly used in any function.

Close the code generation report.

```
rtwdemoclean;
cd(currentDir)
```

See Also

"Optimize global data access" (Simulink)

Related Examples

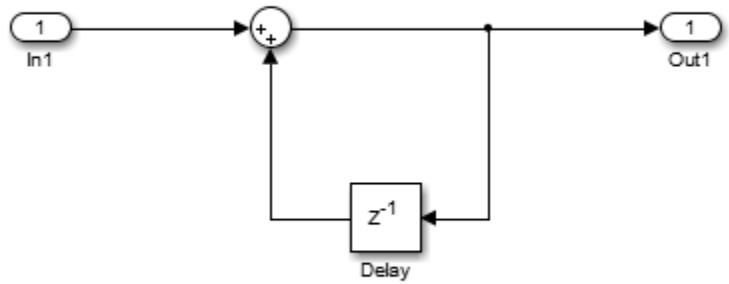
- "Optimization Tools and Techniques" on page 53-7
- "Minimize Computations and Storage for Intermediate Results at Block Outputs" on page 53-36
- "Customize Stack Space Allocation" on page 53-91
- "Reuse Global Block Outputs in the Generated Code" on page 55-14

Reuse Global Block Outputs in the Generated Code

Reduce ROM and RAM consumption and data copies and increase execution speed of generated code. Configure the code generator to reuse global variables by selecting the model configuration parameter **Reuse global block outputs**.

Example

In the Command Window, type rtwdemo_reuse_global.



Generate Code without Optimization

1. In the Configuration Parameters dialog box, on the **All Parameters** tab, verify that **Signal storage reuse** is selected.
2. Clear **Reuse global block outputs** and click **Apply**.
3. On the **Code Generation > Report** pane, select **Static code metrics**.
4. In your system's temporary folder, create a folder for the build and inspection process.

Press **Ctrl+B** to generate code.

```
### Starting build procedure for model: rtwdemo_reuse_global
```

```
### Successful completion of build procedure for model: rtwdemo_reuse_global
```

View the generated code without the optimization. Here is a portion of `rtwdemo_reuse_global.c`.

```
/* Model step function */
void rtwdemo_reuse_global_step(void)
{
  /* Sum: '<Root>/Sum' incorporates:
   *  Inport: '<Root>/In1'
   */
  rtDW.Delay_DSTATE += rtU.In1;

  /* Outport: '<Root>/Out1' */
  rtY.Out1 = rtDW.Delay_DSTATE;
}
```

The generated code contains a data copy to the global variable `rtDW.Delay_DSTATE`. Open the Static Code Metrics Report. The total number of reads and writes for global variables is 8. The total size is 32 bytes.

Enable Optimization and Generate Code

1. On the **All Parameters** tab, select **Reuse global block outputs** and click **Apply**.
2. Generate code.
3. View the generated code with the optimization. Here is a portion of `rtwdemo_reuse_global.c`.

```
### Starting build procedure for model: rtwdemo_reuse_global
### Successful completion of build procedure for model: rtwdemo_reuse_global

/* Model step function */
void rtwdemo_reuse_global_step(void)
{
  /* Sum: '<Root>/Sum' incorporates:
   *  Inport: '<Root>/In1'
   */
  rtY.Out1 += rtU.In1;
}
```

The code generator eliminates a data copy, reduces two statements to one statement and three global variables to two global variables.

Open the Static Code Metrics Report. For global variables, this optimization reduces the total number of reads and writes for global variables from 8 to 5 and the total size from 32 bytes to 24 bytes.

See Also

"Reuse global block outputs" (Simulink)

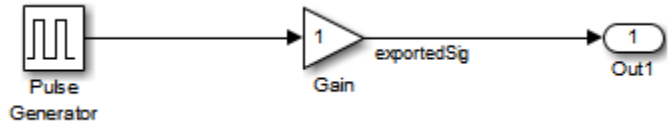
Related Examples

- "Optimization Tools and Techniques" on page 53-7
- "Minimize Computations and Storage for Intermediate Results at Block Outputs" on page 53-36
- "Optimize Global Variable Usage" on page 55-2

Virtualized Output Ports Optimization

The *virtualized output ports* optimization lets you store the signal entering the root output port as a global variable. Clearing the **MAT-file logging** option and setting the TLC variable `FullRootOutputVector` to `0`, both defaults for Embedded Coder, eliminate code and data storage associated with root output ports.

Consider the model in the following block diagram. 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`. The code generator does not generate code or data for the `Out1` block, which has become a virtual block.

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

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

The following code fragment was generated 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 `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;
```

Data maintenance in the external outputs vector can be significant for smaller models that perform benchmarks.

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`. Add the statement

```
%assign FullRootOutputVector = 1
```

to the Embedded Coder system target file. Alternatively, you can enter the assignment from the MATLAB command line using the `set_param` command, the model parameter `TLCOptions`, and the TLC option `-a`. For more information, see "Specify TLC for Code Generation" (Simulink Coder) and "Configure TLC" (Simulink Coder).

For more information on how to control signal storage in generated code, see "Signal Representation in Generated Code" (Simulink Coder).

Related Examples

- "Optimization Tools and Techniques" on page 53-7
- "Specify Buffer Reuse for Multiple Signals in a Path" on page 55-19
- "Optimize Global Variable Usage" on page 55-2

Specify Buffer Reuse for Multiple Signals in a Path

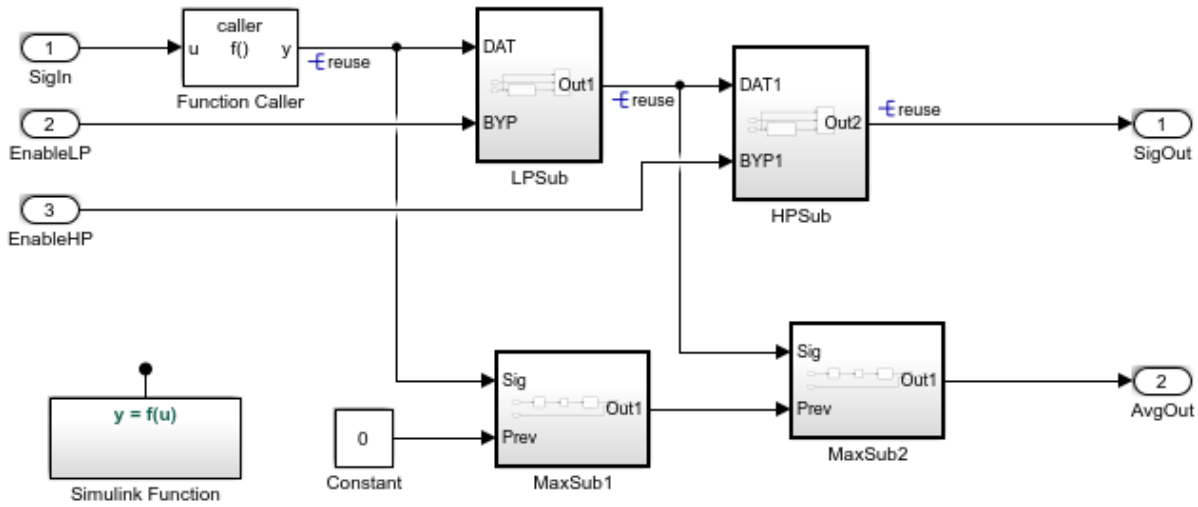
If your model has the optimal parameter settings for removing data copies, you may be able to remove additional data copies by using Simulink.Signal objects to specify buffer reuse. You specify signal objects on signal lines after studying the generated code and the Static Code Metrics Report and identifying areas where you think buffer reuse is possible.

You can specify buffer reuse for multiple signals in a path that can include the root inport and outport signals. You can also specify reuse on just a pair of root inport and outport signals. This optimization reduces ROM and RAM consumption because there are less global variables and data copies in the generated code. Code execution speed also increases.

Example Model

The model rtwdemo_reusable_csc_scheduling contains the nonreusable subsystems `LPSub` and `HPSub` and the reusable subsystems `MaxSub1` and `MaxSub2`.

```
model ='rtwdemo_reusable_csc_scheduling';
open_system(model);
set_param(model,'IncludeMdlTerminateFcn', 'on');
```



Specify a Simulink Signal Object for Reuse

1. Open the base workspace. The Simulink signal object `reuse` is for specifying which buffers to reuse in the generated code. To use a Simulink signal object for buffer reuse, the object must have a **Storage class** of `Reusable (Custom)`.
2. Right-click one of the signal lines that has the signal name `reuse`.
3. Specify a name for the **Signal name** parameter and select the checkbox **Signal name must resolve to Simulink signal object**.

Generate Code

Build the model.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
rtwbuild(model);
```

```
### Starting build procedure for model: rtwdemo_reusable_csc_scheduling
### Successful completion of build procedure for model: rtwdemo_reusable_csc_scheduling
```

The `rtwdemo_reusable_csc_rescheduling.c` file contains this global variable for buffer reuse.

```
/* Definition for custom storage class: Reusable */

real_T reuse[256];
```

The `rtwdemo_reusable_csc_scheduling_step` function contains this code.

```
cfile = fullfile(cgDir,'rtwdemo_reusable_csc_scheduling_ert_rtw','rtwdemo_reusable_csc_
rtwdemodbtype(cfile,'/* Model step function', '/* Model initialize function ', 1, 0);
```

```
/* Model step function */
void rtwdemo_reusable_csc_scheduling_step(void)
{
  /* FunctionCaller: '<Root>/Function Caller' incorporates:
   *  Inport: '<Root>/SigIn'
   */
  f(rtU.SigIn, (&(reuse[0])));

  /* Outputs for Atomic SubSystem: '<Root>/MaxSub1' */
```

```
  /* Constant: '<Root>/Constant' incorporates:
   *  Outport: '<Root>/AvgOut'
   */
  MaxSub1((&(reuse[0])), 0.0, rtY.AvgOut);

  /* End of Outputs for SubSystem: '<Root>/MaxSub1' */

  /* Outputs for Atomic SubSystem: '<Root>/LPSub' */
  LPSub();

  /* End of Outputs for SubSystem: '<Root>/LPSub' */

  /* Outputs for Atomic SubSystem: '<Root>/MaxSub2' */

  /* Outport: '<Root>/AvgOut' */
  MaxSub2((&(reuse[0])), rtY.AvgOut, rtY.AvgOut);

  /* End of Outputs for SubSystem: '<Root>/MaxSub2' */

  /* Outputs for Atomic SubSystem: '<Root>/HPSub' */
  HPSub();

  /* End of Outputs for SubSystem: '<Root>/HPSub' */
}
```

All five functions in the `rtwdemo_reusable_csc_scheduling_step` function can reuse the variable `reuse` because calls to functions `MaxSub1` and `MaxSub2` happen before calls to function `LPSub` and `HPSub`, respectively.

```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

Buffer Reuse for Unit Delay and Delay Blocks

To reuse the signal of a Unit Delay or Delay block

1. Use the same reusable custom storage class specification for a pair of input and state arguments or a pair of output and state arguments of a Unit Delay block or a Delay block.
2. Open the Unit Delay or Delay block parameters dialog box.
3. On the **State Attributes** tab, set the **State name** parameter to the name of the `Simulink.signal` that you want to reuse.

4 On the **State Attributes** tab, select **Signal name must resolve to Simulink signal object**.

5 On the **Main** tab, the **Delay length** parameter must have a value of `1`. The **Initial condition > Source parameter** must be set to `Dialog`.

Limitations for Root Inport and Outport Signals

The following limitations apply to a model in which you specify buffer reuse for a pair of root inport and outport signals.

- The output ports cannot be conditional.
- If the code generator cannot reuse the same buffer in a top model, the generated code contains additional buffers. If the same model is a reference model, the code generator reports an error. To resolve the error, remove the Simulink.signal specification from the signal that connects to the outport port.
- When you run an executable produced by code generation, and you reuse a pair of root inport and outport signals, when the root input value is zero, the root output value must also be zero. If the output value is nonzero and you reuse the signals, then the results from the simulation can differ from the results produced by the executable.

Limitations for the Model

The following limitations apply to a model in which you specify buffer reuse for signals.

- Signals that you specify for reuse must have the same data types and sampling rates.
- For user-specified buffer reuse, blocks that modify a signal specified for reuse must execute before blocks that use the original signal value. Sometimes the code generator has to change the block operation order so that buffer reuse can occur. For models in which the code generator is unable to reorder block operations, buffer reuse does not occur.
- For models in which the code generator reorders block operations so that `Simulink.Signal` reuse can occur, you can observe the difference in the sorted order. In the model window, select **Display > Blocks > Sorted Execution Order**. To display the sorted execution order during simulation, select **Simulation > Update Diagram**. To display the execution order in the generated code, select **Code > C/C++ Code > Build Model**.

Related Examples

- “Optimization Tools and Techniques” on page 53-7

- "Control Signals and States in Code by Applying Storage Classes" (Simulink Coder)
- "Simulink Package Custom Storage Classes" on page 23-5
- "Virtualized Output Ports Optimization" on page 55-17
- "Design Data Interface by Configuring Inport and Outport Blocks" on page 19-134

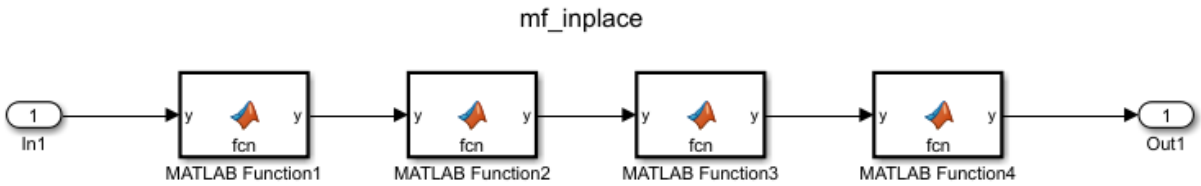
Specify Buffer Reuse for MATLAB Function Blocks in a Path

In this section...

You can specify buffer reuse across MATLAB Function blocks by using the same variable name for the input and output arguments. The code generator tries to reuse the output of one MATLAB Function block as the input to the next MATLAB Function block. This optimization conserves RAM and ROM consumption and reduces data copies.

Example Model

1 Use Inport, Outport, and MATLAB Function blocks to create the model `mf_inplace`.



2 Open each MATLAB Function block and copy the following code:

```
function y = fcn(y)
%#codegen

y=y+4;
```

3 Open the Configuration Parameters dialog box. On the **Code Generation** tab, change the **System target file** to `ert.tlc`.

4 On the **Solver** tab, change the **Type** parameter to `Fixed-step`.

Generate Code with Optimization

Generate code for the model. The `mf_inplace.c` file contains this code:

```
void mf_inplace_MATLABFunction(real_T *rty_y)
```

```
{
  *rty_y += 4.0;
}

void mf_inplace_step(void)
{
  real_T rtb_y_p5;
  rtb_y_p5 = mf_inplace_U.In1;
  mf_inplace_MATLABFunction(&rtb_y_p5);
  mf_inplace_MATLABFunction(&rtb_y_p5);
  mf_inplace_MATLABFunction(&rtb_y_p5);
  mf_inplace_Y.Out1 = rtb_y_p5;
  mf_inplace_MATLABFunction(&mf_inplace_Y.Out1);
}
```

The code generator reuses the variable `rtb_y_p5` for the input and output arguments of each MATLAB Function block.

Note: On the **Code Generation** tab in the Subsystem Block Parameters dialog box, if the **Function packaging** parameter is set to `Nonreusable function` and the **Function interface** parameter is set to `Allow arguments`, the code generator cannot reuse the input and output arguments.

Related Examples

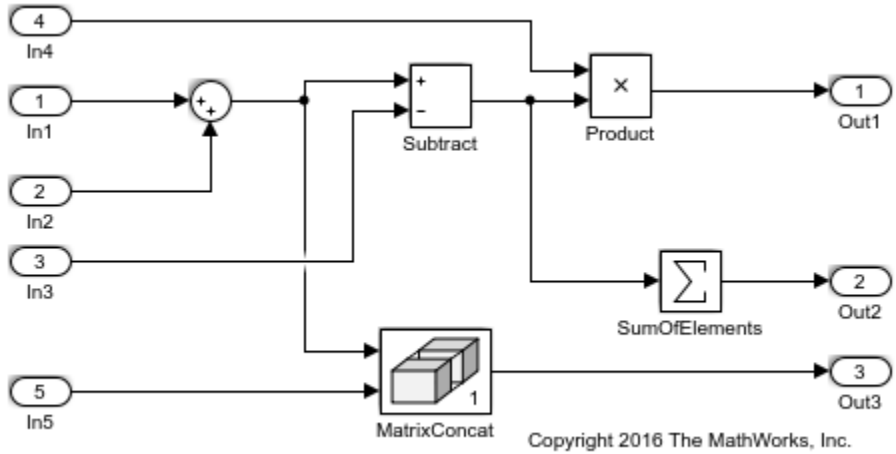
- "What Is a MATLAB Function Block?" (Simulink)
- "Specify Buffer Reuse for Multiple Signals in a Path" on page 55-19

Remove Data Copies by Reordering Block Operations in the Generated Code

This example shows how to remove data copies by reordering block operations in the generated code. This optimization conserves RAM and ROM consumption and improves code execution speed.

Example Model

In the model matlab:rtwdemo_optimizeblockorder, the signal that leaves the Sum block enters a Subtract block and a Concatenate block. The signal that leaves the Subtract block enters a Product block and a Sum of Elements block.

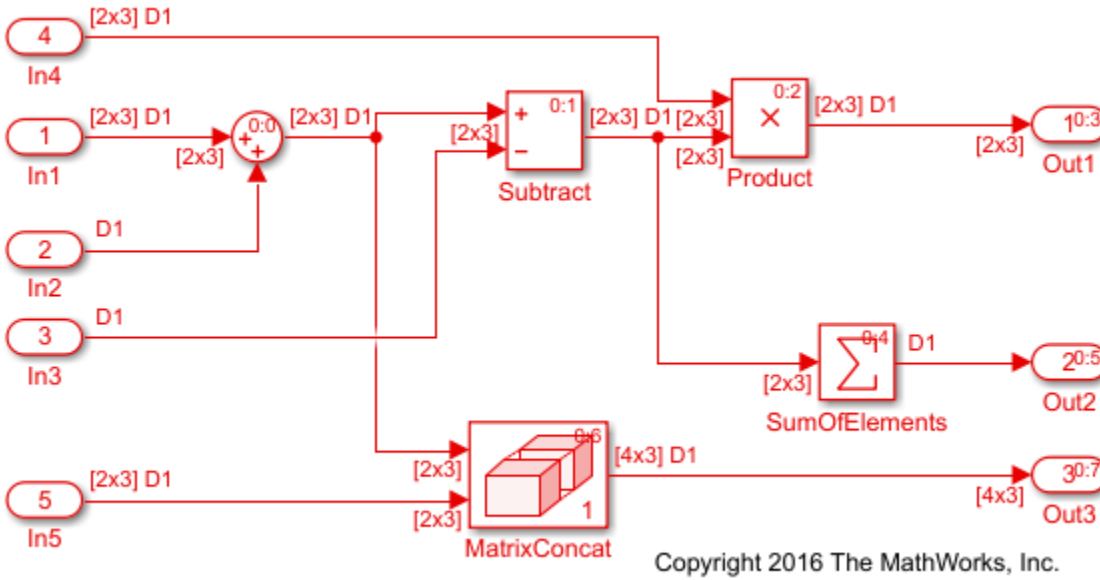


Generate Code without Optimization

In your system's temporary folder, create a folder for the build and inspection process and build the model.

```
### Starting build procedure for model: rtwdemo_optimizeblockorder
### Successful completion of build procedure for model: rtwdemo_optimizeblockorder
```

The image shows the default block order in the generated code. The Subtract block executes before the Concatenate block. The Product block executes before the Sum of Elements block.



View the generated code without the optimization. Here is `rtwdemo_optimizeblockorder.c`.

```
/* Model step function */
void rtwdemo_optimizeblockorder_step(void)
{
  real_T rtb_Sum2x3[6];
  int32_T i;
  real_T rtb_Sum2x3_d;
  real_T rtb_Subtract;

  /* Sum: '<Root>/SumOfElements' */
  rtY.Out2 = -0.0;
  for (i = 0; i < 6; i++) {
    /* Sum: '<Root>/Sum2x3' incorporates:
     *  Inport: '<Root>/In1'
     *  Inport: '<Root>/In2'
     */
    rtb_Sum2x3_d = rtU.In1[i] + rtU.In2;
```

```
    /* Sum: '<Root>/Subtract' incorporates:
     *  Inport: '<Root>/In3'
     */
    rtb_Subtract = rtb_Sum2x3_d - rtU.In3;

    /* Outport: '<Root>/Out1' incorporates:
     *  Inport: '<Root>/In4'
     *  Product: '<Root>/Product'
     */
    rtY.Out1[i] = rtU.In4[i] * rtb_Subtract;

    /* Sum: '<Root>/Sum2x3' */
    rtb_Sum2x3[i] = rtb_Sum2x3_d;

    /* Sum: '<Root>/SumOfElements' */
    rtY.Out2 += rtb_Subtract;
  }

  /* Concatenate: '<Root>/MatrixConcat ' */
  for (i = 0; i < 3; i++) {
    /* Outport: '<Root>/Out3' incorporates:
     *  Inport: '<Root>/In5'
     */
    rtY.Out3[i << 2] = rtb_Sum2x3[i << 1];
    rtY.Out3[2 + (i << 2)] = rtU.In5[i << 1];
    rtY.Out3[1 + (i << 2)] = rtb_Sum2x3[(i << 1) + 1];
    rtY.Out3[3 + (i << 2)] = rtU.In5[(i << 1) + 1];
  }

  /* End of Concatenate: '<Root>/MatrixConcat ' */
}
```

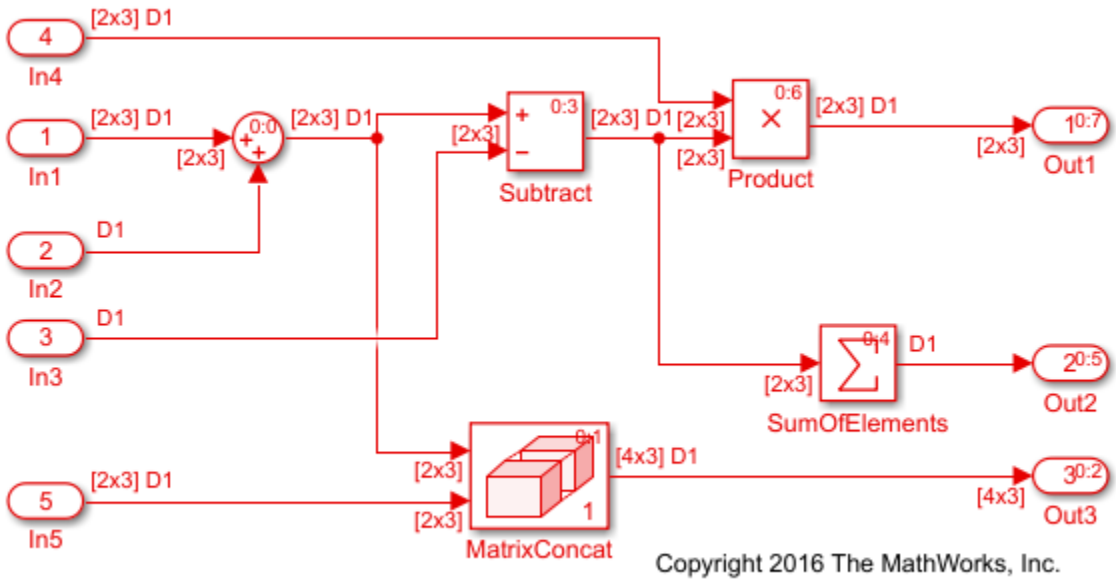
With the default order, the generated code contains three buffers, `rtb_Sum2x3[6]`, `rtb_Sum2x3_d`, and `rtb_Subtract`. The generated code contains these temporary variables and associated data copies because the Matrix Concatenate block must use the output from the Sum block and the Sum of Elements block must use the output from the Subtract block.

Generate Code with Optimization

1. In the Configuration Parameters dialog box, change the **Optimize block order in the generated code** parameter to `Improved Execution Speed`.
2. Generate code for the model.

```
### Starting build procedure for model: rtwdemo_optimizeblockorder
### Successful completion of build procedure for model: rtwdemo_optimizeblockorder
```

The image shows the optimized block order in the generated code. The Subtract block executes after the Concatenate block. The Product block executes after the Sum of Elements block.



View the generated code with the optimization. Here is `rtwdemo_optimizeblockorder.c`.

```
/* Model step function */
void rtwdemo_optimizeblockorder_step(void)
{
  int32_T i;

  /* Sum: '<Root>/Sum2x3' incorporates:
   *  Inport: '<Root>/In1'
   *  Inport: '<Root>/In2'
   */
```

```
  for (i = 0; i < 6; i++) {
    rtY.Out1[i] = rtU.In1[i] + rtU.In2;
  }

  /* End of Sum: '<Root>/Sum2x3' */

  /* Concatenate: '<Root>/MatrixConcat ' */
  for (i = 0; i < 3; i++) {
    /* Outport: '<Root>/Out3' incorporates:
     *  Inport: '<Root>/In5'
     */
    rtY.Out3[i << 2] = rtY.Out1[i << 1];
    rtY.Out3[2 + (i << 2)] = rtU.In5[i << 1];
    rtY.Out3[1 + (i << 2)] = rtY.Out1[(i << 1) + 1];
    rtY.Out3[3 + (i << 2)] = rtU.In5[(i << 1) + 1];
  }

  /* End of Concatenate: '<Root>/MatrixConcat ' */

  /* Sum: '<Root>/SumOfElements' */
  rtY.Out2 = -0.0;
  for (i = 0; i < 6; i++) {
    /* Sum: '<Root>/Subtract' incorporates:
     *  Inport: '<Root>/In3'
     */
    rtY.Out1[i] -= rtU.In3;

    /* Sum: '<Root>/SumOfElements' */
    rtY.Out2 += rtY.Out1[i];

    /* Outport: '<Root>/Out1' incorporates:
     *  Inport: '<Root>/In4'
     *  Product: '<Root>/Product'
     */
    rtY.Out1[i] *= rtU.In4[i];
  }
}
```

In the optimized code, the three buffers `rtb_Sum2x3[6]`, `rtb_Sum2x3_d`, and `rtb_Subtract` and their associated data copies are gone. The generated code does not require these temporary variables to hold the outputs of the Sum and Subtract blocks because the Subtract block executes after the Concatenate block and the Product block executes after the Sum of Elements block.

To implement buffer reuse, the code generator does not violate user-specified block priorities.

See Also

"Optimize block operation order in the generated code" (Simulink)

Related Examples

- "Data Copy Reduction"

Execution Speed in Embedded Coder

Reduce Memory Requirements for Signals

To reduce the memory requirements of your real-time program, select the configuration parameter **Signal storage reuse**. Selecting **Signal storage reuse** enables parameters that provide the capability to reuse memory allocated for signals: **Reuse local block outputs**, **Reuse global block outputs**, and **Optimize global data access**. Clearing **Signal storage reuse** makes all block outputs global and unique, which often significantly increases RAM and ROM usage.

When you select **Reuse local block outputs**, the code generator reuses local (function) variables for block outputs wherever possible. When you select **Reuse global block outputs**, the code generator reuses global (function) variables wherever possible.

The **Optimize global data access** parameter has the following settings `None`, `Use global to hold temporary results`, and `Minimize global data access`. When you select `None`, the code generator uses the default optimizations. The setting `Use global to hold temporary results` maximizes the use of global variables. The setting `Minimize global data access` minimizes the use of global variables by using local variables to hold intermediate values.

Related Examples

- "Enable and Reuse Local Block Outputs in Generated Code" (Simulink Coder)
- "Reuse Global Block Outputs in the Generated Code" on page 55-14
- "Optimize Global Variable Usage" on page 55-2

Remove Initialization Code

Consider selecting the “Remove root level I/O zero initialization” (Simulink) and “Remove internal data zero initialization” (Simulink) options on the **Optimization > General** 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 your application might not require it. Many embedded application environments initialize RAM to zero at startup, making generation of initialization code redundant.

However, be aware that if you select **Remove internal data zero initialization**, memory might not be 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 data zero initialization** if you want to test the behavior of your design during a warm boot (that is, a restart without full system reinitialization).

In cases where you have turned on **Remove internal data zero initialization** but still want to get the same answer on every run from an S-function produced by the code generator, 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” (Simulink).

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

Note also that data of `ImportedExtern` or `ImportedExternPointer` storage classes are not initialized, regardless of the settings of these options.

Related Examples

- "Optimization Tools and Techniques" on page 53-7
- "Eliminate Zero Initialization Code for Internal Data" on page 56-5
- "Remove Initialization Code for Root-Level Inports and Outports Set to Zero" on page 56-28
- "Optimize Generated Code Using `memset` Function" on page 53-93

Eliminate Zero Initialization Code for Internal Data

This example shows how to eliminate generated code that initializes internal data with zeroes, for example global DWork vectors, to reduce the size of the code and to accelerate model initialization.

Overview

During model initialization, generated code can initialize internal data by using assignments to zero. DWork vectors are an example of internal data.

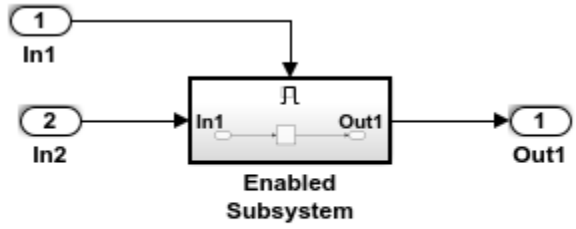
If the data are global variables in the generated code, and if the target environment already initializes global variables with zeroes, you can remove the corresponding lines of model initialization code.

This optimization removes unnecessary zero initialization code, providing these benefits:

- Reduction in size of generated code
- Increased execution speed of generated code

Open Example Model

Open the model rtwdemo_internal_init. The model contains an enabled subsystem whose initial output is zero. The subsystem contains a Unit Delay block whose initial condition is 0.



Generate Code Without Optimization

Build the model using Embedded Coder.

```
### Starting build procedure for model: rtwdemo_internal_init
### Successful completion of build procedure for model: rtwdemo_internal_init
```

View the following code from the generated file `rtwdemo_internal_init.c`.

```
/* Model initialize function */
void rtwdemo_internal_init_initialize(void)
{
  /* Registration code */

  /* initialize error status */
  rtmSetErrorStatus(rtM, (NULL));

  /* states (dwork) */
  (void) memset((void *)&rtDWork, 0,
                sizeof(D_Work));

  /* SystemInitialize for Enabled SubSystem: '<Root>/Enabled Subsystem' */
  /* InitializeConditions for UnitDelay: '<S1>/Unit Delay' */
  rtDWork.UnitDelay_DSTATE = 0.0;

  /* End of SystemInitialize for SubSystem: '<Root>/Enabled Subsystem' */
}

/*
```

Enable Optimization

Open the Configuration Parameters dialog box. On the **Optimization** pane, select **Remove internal data zero initialization**.

Alternatively, you can use the command prompt to enable the optimization. To enable the optimization, set the model parameter `ZeroInternalMemoryAtStartup` to `'off'`.

```
set_param(model, 'ZeroInternalMemoryAtStartup', 'off');
```

Generate Code with Optimization

Build the model using Embedded Coder.

```
### Starting build procedure for model: rtwdemo_internal_init
### Successful completion of build procedure for model: rtwdemo_internal_init
```

View the following code from the file `rtwdemo_internal_init.c`. The generated code does not initialize internal data by assignment to zero.

```
/* Model initialize function */
void rtwdemo_internal_init_initialize(void)
{
  /* (no initialization code required) */
}

/*
```

See Also

"Remove internal data zero initialization" (Simulink)

Related Examples

- "Optimization Tools and Techniques" on page 53-7
- "Remove Initialization Code" on page 56-3

Generate Pure Integer Code If Possible

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

See Also

"Support: floating-point numbers" (Simulink Coder)

Related Examples

- "Optimization Tools and Techniques" on page 53-7
- "Data Types Supported by Simulink" (Simulink)

Disable MAT-File Logging

Disable MAT-file logging by clearing **Configuration Parameters > All Parameters > MAT-file logging**. 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** parameter lets you exploit further efficiencies under certain conditions. See "Virtualized Output Ports Optimization" on page 55-17 for information.

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

See Also

"MAT-file logging" (Simulink Coder)

Related Examples

- "Optimization Tools and Techniques" on page 53-7
- "Virtualized Output Ports Optimization" on page 55-17

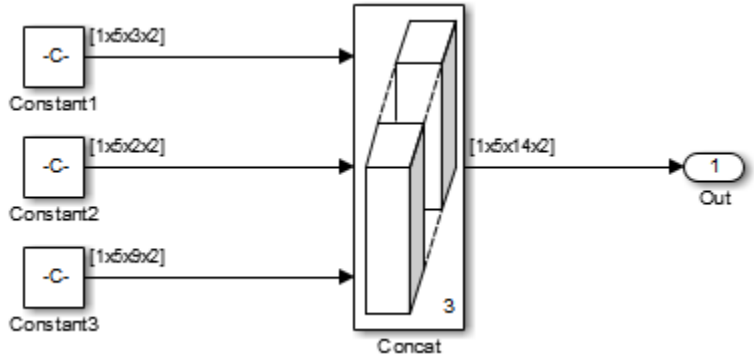
Simplify Multiply Operations in Array Indexing

In this section...

The generated code might have multiply operations when indexing an element of an array. You can select the optimization parameter “Simplify array indexing” (Simulink) to replace multiply operations in the array index with a temporary variable. This optimization can improve execution speed by reducing the number of times the multiply operation executes.

Example Model

If you have the following model:



The Constant blocks have the following **Constant value**:

- `Const1`: `reshape(1:30,[1 5 3 2])`
- `Const2`: `reshape(1:20,[1 5 2 2])`
- `Const3`: `reshape(1:90,[1 5 9 2])`

The Concatenate block parameter **Mode** is set to `Multidimensional array`. The Constant blocks **Sample time** parameter is set to –1.

Generate Code

Building the model with the **Simplify array indexing** parameter turned off generates the following code:

```
int32_T i;
int32_T i_0;
int32_T i_1;

for (i = 0; i < 2; i++) {
  for (i_1 = 0; i_1 < 3; i_1++) {
    for (i_0 = 0; i_0 < 5; i_0++) {
      ex_arrayindex_Y.Out[(i_0 + 5 * i_1) + 70 * i] =
        ex_arrayindex_P.Constant1_Value[(5 * i_1 + i_0) + 15 * i];
    }
  }
}

for (i = 0; i < 2; i++) {
  for (i_1 = 0; i_1 < 2; i_1++) {
    for (i_0 = 0; i_0 < 5; i_0++) {
      ex_arrayindex_Y.Out[(i_0 + 5 * (i_1 + 3)) + 70 * i] =
        ex_arrayindex_P.Constant2_Value[(5 * i_1 + i_0) + 10 * i];
    }
  }
}

for (i = 0; i < 2; i++) {
  for (i_1 = 0; i_1 < 9; i_1++) {
    for (i_0 = 0; i_0 < 5; i_0++) {
      ex_arrayindex_Y.Out[(i_0 + 5 * (i_1 + 5)) + 70 * i] =
        ex_arrayindex_P.Constant3_Value[(5 * i_1 + i_0) + 45 * i];
    }
  }
}
```

Generate Code with Optimization

Open the Configuration Parameters dialog box and select the **Simplify array indexing** parameter on the **All Parameters** tab. Build the model again. In the generated code,

`[(i_0 + tmp_1) + tmp]` replaces a multiply operation in the array index, `[(i_0 + 5 * i_1) + 70 * i]`. The generated code is now:

```
int32_T i;
int32_T i_0;
int32_T i_1;
int32_T tmp;
int32_T tmp_0;
int32_T tmp_1;

tmp = 0;
tmp_0 = 0;
for (i = 0; i < 2; i++) {
  tmp_1 = 0;
  for (i_1 = 0; i_1 < 3; i_1++) {
    for (i_0 = 0; i_0 < 5; i_0++) {
      ex_arrayindex_Y.Out[(i_0 + tmp_1) + tmp] =
        ex_arrayindex_P.Constant1_Value[(i_0 + tmp_1) + tmp_0];
    }

    tmp_1 += 5;
  }

  tmp += 70;
  tmp_0 += 15;
}

tmp = 0;
tmp_0 = 0;
for (i = 0; i < 2; i++) {
  tmp_1 = 0;
  for (i_1 = 0; i_1 < 2; i_1++) {
    for (i_0 = 0; i_0 < 5; i_0++) {
      ex_arrayindex_Y.Out[((i_0 + tmp_1) + tmp) + 15] =
        ex_arrayindex_P.Constant2_Value[(i_0 + tmp_1) + tmp_0];
    }

    tmp_1 += 5;
  }

  tmp += 70;
  tmp_0 += 10;
}
```

```
tmp = 0;
tmp_0 = 0;
for (i = 0; i < 2; i++) {
  tmp_1 = 0;
  for (i_1 = 0; i_1 < 9; i_1++) {
    for (i_0 = 0; i_0 < 5; i_0++) {
      ex_arrayindex_Y.Out[((i_0 + tmp_1) + tmp) + 25] =
        ex_arrayindex_P.Constant3_Value[(i_0 + tmp_1) + tmp_0];
    }

    tmp_1 += 5;
  }

  tmp += 70;
  tmp_0 += 45;
}
```

See Also

"Simplify array indexing" (Simulink)

Related Examples

- "Optimization Tools and Techniques" on page 53-7
- "Use `memcpy` Function to Optimize Generated Code for Vector Assignments" on page 53-52
- "Vector Operation Optimization" on page 53-97

Replace `boolean` with Specific Integer Data Type

Depending on the architecture of the processor that your production hardware uses, you can improve the execution speed of generated code. Select a specific integer data type to use for the built-in type `boolean`. Using data type replacement, in the generated code you can replace the `boolean` built-in data type with one of these integer types:

- `int8`
- `uint8`
- `int`*`n`*

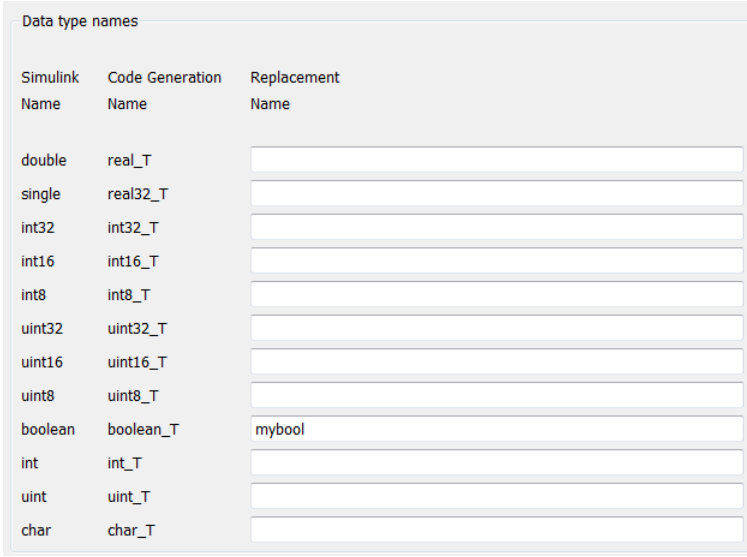
Replace *n* with `8`, `16`, or `32` to match the integer word size for the production hardware.

This example shows how to replace the data type `boolean` with the integer data type `int32` in the code generated for a 32-bit hardware target.

1. Define a `Simulink.AliasType` object with a base type of `int32`. Name the object using the replacement name that you want to appear in the generated code.

   ```
   mybool = Simulink.AliasType;
   mybool.BaseType = 'int32';
   ```

2. Open an ERT-based model. In the Configuration Parameters dialog box **Data Type Replacement** pane, specify the **Replacement Name** field for the data type `boolean` as `mybool`.



View the generated file `rtwtypes.h`. The code maps the identifier `mybool` to the native integer type of the target hardware by creating `typedef` statements.

```
/* Generic type definitions ... */
...
typedef int boolean_T;
 ...
/* Define Simulink Coder replacement data types. */
typedef boolean_T mybool;    /* User defined replacement datatype for boolean_T */
```

View the generated file *`model`*`.c`. The code declares Boolean variables using the type `mybool`. For example, if the model has a Boolean output `Out1`, the generated code declares the corresponding variable using `mybool`.

```
  mybool Out1;                     /* '<Root>/Out1' */
```

See Also

`Simulink.AliasType`

Related Examples

- “Data Type Replacement” on page 21-36

More About

- "What Are User-Defined Data Types?" on page 21-2

Remove Code That Guards Against Division Exceptions for Integers and Fixed-Point Data

Optimize generated code by removing code that protects against division by zero and overflows in division `INT_MIN/-1` operations for integers and fixed-point data. If you are sure that these arithmetic exceptions do not occur during program execution, enable this optimization.

This optimization:

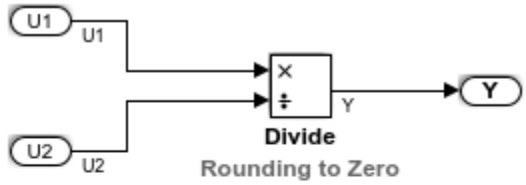
- Increases execution speed.
- Reduces ROM consumption.

NOTE: If you enable this optimization, it is possible that simulation results and results from generated code are not in bit-for-bit agreement. This example requires an Embedded Coder® license.

Example Model

In the model rtwdemo_nzcheck, two signals of type `int8` feed into a divide block.

```
model = 'rtwdemo_nzcheck';
open_system(model);
```



Generate Code

In your system's temporary folder, create a temporary folder for the build and inspection process.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
```

Build the model.

```
set_param(model, 'NoFixptDivByZeroProtection', 'off');
rtwbuild(model);
```

```
### Starting build procedure for model: rtwdemo_nzcheck
### Successful completion of code generation for model: rtwdemo_nzcheck
```

View the generated code without the optimization. Here is a portion of `rtwdemo_nzcheck.c`.

```
cfile = fullfile(cgDir,'rtwdemo_nzcheck_ert_rtw','rtwdemo_nzcheck.c');
rtwdemodbtype(cfile,'/* Real-time model','/* Model step function',1, 1);
```

```
/* Real-time model */
RT_MODEL_rtwdemo_nzcheck rtwdemo_nzcheck_M_;
RT_MODEL_rtwdemo_nzcheck *const rtwdemo_nzcheck_M = &rtwdemo_nzcheck_M_;
int32_T div_s32(int32_T numerator, int32_T denominator)
{
  int32_T quotient;
  uint32_T tempAbsQuotient;
  if (denominator == 0) {
    quotient = numerator >= 0 ? MAX_int32_T : MIN_int32_T;

    /* Divide by zero handler */
  } else {
    tempAbsQuotient = (numerator < 0 ? ~(uint32_T)numerator + 1U : (uint32_T)
                       numerator) / (denominator < 0 ? ~(uint32_T)denominator +
      1U : (uint32_T)denominator);
    quotient = (numerator < 0) != (denominator < 0) ? -(int32_T)tempAbsQuotient :
      (int32_T)tempAbsQuotient;
  }

  return quotient;
}
```

Enable Optimization

1. Open the Configuration Parameters dialog box.
2. On the **Optimization** pane, select **Remove code that protects against division arithmetic exceptions**.

Alternatively, you may use the command-line API to enable the optimization:

```
set_param(model, 'NoFixptDivByZeroProtection', 'on');
```

Generate Code with Optimization

The optimized code does not contain code that checks for whether or not the divisor has a value of zero.

Build the model.

```
rtwbuild(model);
```

```
### Starting build procedure for model: rtwdemo_nzcheck
### Successful completion of code generation for model: rtwdemo_nzcheck
```

The following is a portion of `rtwdemo_nzcheck.c`. The code that protects against division arithmetic exceptions is not in the generated code.

```
rtwdemodbtype(cfile,'/* Real-time model','/* Model step function',1, 1);
```

```
/* Real-time model */
RT_MODEL_rtwdemo_nzcheck rtwdemo_nzcheck_M_;
RT_MODEL_rtwdemo_nzcheck *const rtwdemo_nzcheck_M = &rtwdemo_nzcheck_M_;
```

Close the model and code generation report.

```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

See Also

"Remove code that protects against division arithmetic exceptions" (Simulink)

Related Examples

- "Optimization Tools and Techniques" on page 53-7
- "Remove Code From Floating-Point to Integer Conversions That Wraps Out-of-Range Values" on page 53-23
- "Remove Code That Maps NaN to Integer Zero" on page 53-26
- "Division Arithmetic Exceptions in Generated Code" on page 56-21

Division Arithmetic Exceptions in Generated Code

The **Remove code that protects against division arithmetic exceptions** parameter in the **Optimization** pane of the Configuration Parameters dialog box controls the generation of code that protects against division arithmetic exceptions in integer and fixed-point operations. Division arithmetic exceptions include division by zero and `INT_MIN/-1`, which results in a quotient that cannot be represented.

When the parameter is selected, the generated code does not contain code that guards against these types of exceptions. This produces smaller, more efficient code, however it can affect numerical results. Select the parameter if you are sure that your model would not encounter these exceptions to optimize the efficiency of the generated code.

When the parameter is not selected, the generated code does not contain code that guards against division arithmetic exceptions. The added protection code checks that there is a numerical match between simulation and code generation for division operations at a cost of code size and performance.

Division by Zero

Division by zero is undefined and results in a runtime error in the generated code. When the **Remove code that protects against division arithmetic exceptions** parameter is selected, the code generator does not produce code that protects against division by zero. Select this option only when you are sure that your model will not produce such a division operation.

INT_MIN/-1

When you divide the minimum representable value of a signed integer by negative one, the ideal result is equal to the maximum representable value plus one (`INT_MAX + 1`), which is not representable. This exception may cause the application to unexpectedly halt or crash at run-time. When the **Remove code that protects against division arithmetic exceptions** parameter is selected, the code generator does not produce code that protects against this situation. Select this option only when you are sure that your model will not produce such a division operation.

The following illustrates an example of this type of exception:

```
a = int32(-2147483648);
b = a/-1
```

The ideal quotient of this operation is `2147483648`, but this is not representable in an `int32` data type, resulting in an exception at runtime.

Other Factors Affecting Generated Code of Division Operations

In addition to the **Remove code that protects against division arithmetic exceptions** parameter, there are several other factors that can affect the appearance of code generated for division operations. The manner in which this parameter controls code generated from blocks containing MATLAB code with integer or fixed-point division operations differs from the built-in Divide block in Simulink. Blocks containing MATLAB code include MATLAB Function blocks and Stateflow charts using MATLAB action language. To balance the efficiency and semantics of fixed-point and integer divisions in these blocks, use `fi` objects and set the `fimath` properties to fit your needs. Usage of `fi` and `fimath` objects requires a Fixed-Point Designer license.

Rounding and overflow modes also affect the size and efficiency of the generated code. For more information, see "Optimize Generated Code with the Model Advisor" (Fixed-Point Designer).

See Also

"Remove code that protects against division arithmetic exceptions" (Simulink)

Related Examples

- "Remove Code That Guards Against Division Exceptions for Integers and Fixed-Point Data" on page 56-17
- "Optimization Tools and Techniques" on page 53-7

Optimize Generated Code by Consolidating Redundant If-Else Statements

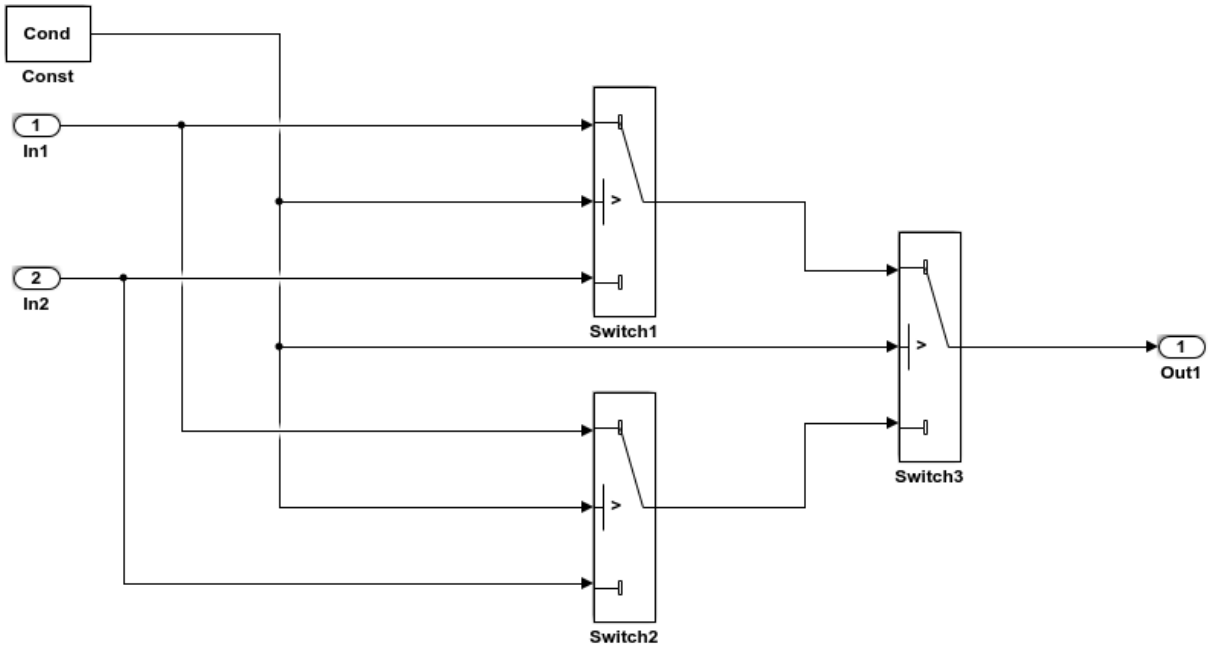
This example shows how to optimize generated code by combining `if-else` statements that share the same condition. This optimization:

- Improves control flow.
- Reduces code size.
- Reduces RAM consumption.
- Increases execution speed.

Example

The model rtwdemo_controlflow_opt contains three Switch blocks. The Constant block provides the control input to the Switch blocks. The variable named `Cond` determines the value of the Constant block.

```
model = 'rtwdemo_controlflow_opt';
open_system(model);
```



Generate Code

Create a temporary folder for the build and inspection process.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
```

Build the model.

```
rtwbuild(model)
```

```
### Starting build procedure for model: rtwdemo_controlflow_opt
### Successful completion of build procedure for model: rtwdemo_controlflow_opt
```

These lines of `rtwdemo_controlflow_opt.c` code show that in the generated code, two `if-else` statements and one `else-if` statement represent the three Switch blocks.

```
cfile = fullfile(cgDir,'rtwdemo_controlflow_opt_ert_rtw',...
    'rtwdemo_controlflow_opt.c');
rtwdemodbtype(cfile,'/* Model step', '/* Model initialize', 1, 0);
```

```
/* Model step function */
void rtwdemo_controlflow_opt_step(void)
{
  /* Switch: '<Root>/Switch3' incorporates:
   *  Constant: '<Root>/Const'
   *  Switch: '<Root>/Switch2'
   */
  if (Cond) {
    /* Switch: '<Root>/Switch1' */
    if (Cond) {
      /* Outport: '<Root>/Out1' incorporates:
       *  Inport: '<Root>/In1'
       */
      rtY.Out1 = rtU.In1;
    } else {
      /* Outport: '<Root>/Out1' incorporates:
       *  Inport: '<Root>/In2'
       */
      rtY.Out1 = rtU.In2;
    }

    /* End of Switch: '<Root>/Switch1' */
  } else if (Cond) {
    /* Switch: '<Root>/Switch2' incorporates:
     *  Inport: '<Root>/In1'
     *  Outport: '<Root>/Out1'
     */
    rtY.Out1 = rtU.In1;
  } else {
    /* Outport: '<Root>/Out1' incorporates:
     *  Inport: '<Root>/In2'
     */
    rtY.Out1 = rtU.In2;
  }

  /* End of Switch: '<Root>/Switch3' */
}
```

Enable Optimization

1. Open the Configuration Parameters dialog box.
2. On the **Code generation**-> **Code Style** pane, clear **Preserve condition expression in if statement**. This parameter is on by default.

Alternatively, use the command-line API to turn off the parameter:

```
set_param(model, 'PreserveIfCondition', 'off');
```

Generate Code with Optimization

In the optimized code, the code generator consolidates the two `if-else` statements and one `else-if` statement into one `if-else` statement. The code generator consolidates these statements because they all share the same condition. There is no intervening code that affects the outcomes of these statements.

Build the model.

```
rtwbuild(model)
```

```
### Starting build procedure for model: rtwdemo_controlflow_opt
### Successful completion of build procedure for model: rtwdemo_controlflow_opt
```

Here is the `rtwdemo_controlflow_opt.c` optimized code.

```
rtwdemodbtype(cfile,'/* Model step', '/* Model initialize', 1, 0);
```

```
/* Model step function */
void rtwdemo_controlflow_opt_step(void)
{
  /* Switch: '<Root>/Switch1' incorporates:
   *  Constant: '<Root>/Const'
   *  Switch: '<Root>/Switch3'
   */
  if (Cond) {
    /* Outport: '<Root>/Out1' incorporates:
     *  Inport: '<Root>/In1'
     */
    rtY.Out1 = rtU.In1;
  } else {
    /* Outport: '<Root>/Out1' incorporates:
     *  Inport: '<Root>/In2'
```

```
     */
    rtY.Out1 = rtU.In2;
  }

  /* End of Switch: '<Root>/Switch1' */
}
```

Close the model and clean up.

```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

See Also

"Preserve condition expression in if statement"

Related Examples

- "Optimization Tools and Techniques" on page 53-7
- "Eliminate Dead Code Paths in Generated Code" on page 53-61

Remove Initialization Code for Root-Level Inports and Outports Set to Zero

This example shows how to optimize generated code by removing initialization code for root-level inports and outports set to zero. If your embedded application does not require generating initialization code for external data whose value is zero, you can enable this optimization. For example, many embedded application environments initialize RAM to zero at startup, making generation of initialization code redundant. This optimization:

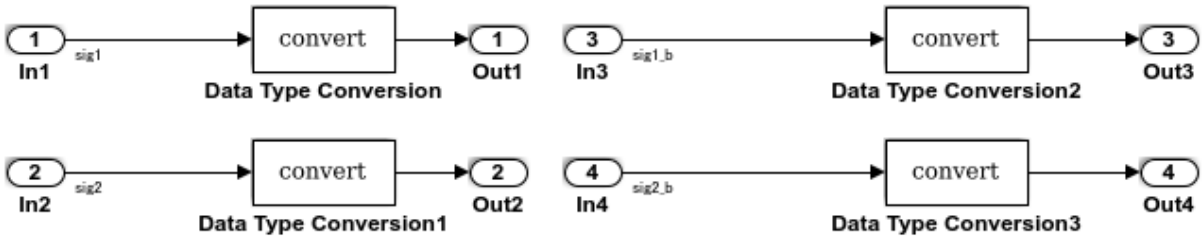
- Increases execution speed.
- Reduces ROM consumption.

Note: This example requires an Embedded Coder® license.

Example

In the model rtwdemo_rootlevel_zero_initialization, all of the input and output signals have a numeric value of zero. Because signals `sig1` and `sig2` have data types `int16` and `Boolean`, respectively, and all of the output signals have data type `double`, these signals also have initial values of bitwise zero. The signals have an integer bit pattern of `0`, meaning that all bits are off. Signals `sig1_b` and `sig2_b` have a fixed-point data type with bias, so their initial value is not bitwise zero.

```
model = 'rtwdemo_rootlevel_zero_initialization';
open_system(model);
```



Generate Code

In your system temporary folder, create a temporary folder for the build and inspection process.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
```

Build the model.

```
set_param(model, 'ZeroExternalMemoryAtStartup','on');
rtwbuild(model)
```

```
### Starting build procedure for model: rtwdemo_rootlevel_zero_initialization
### Successful completion of build procedure for model: rtwdemo_rootlevel_zero_initiali
```

These lines of `rtwdemo_rootlevel_zero_initialization.c` code show the initialization of root-level inports and outports without the optimization. The four input signals are individually initialized as global variables. The four output signals are members of a global structure that the `memset` function initializes to bitwise zero.

```
cfile = fullfile(cgDir,'rtwdemo_rootlevel_zero_initialization_ert_rtw',...
    'rtwdemo_rootlevel_zero_initialization.c');
rtwdemodbtype(cfile, 'rtwdemo_rootlevel_zero_initialization_initialize',...
    'trailer for generated code', 1, 0);
```

```
void rtwdemo_rootlevel_zero_initialization_initialize(void)
{
  /* Registration code */

  /* external inputs */
  sig1 = 0;
  sig2 = false;
  sig1_b = -3;
  sig2_b = -3;

  /* external outputs */
  (void) memset((void *)&rtY, 0,
                sizeof(ExternalOutputs));
}

/*
```

Enable Optimization

1. Open the Configuration Parameters dialog box.
2. On the **Optimization** pane, select **Remove root level I/O zero initialization**.

Alternatively, use the command-line API to enable the optimization:

```
set_param(model, 'ZeroExternalMemoryAtStartup','off');
```

Generate Code with Optimization

The optimized code does not contain initialization code for the input signals `sig1`, `sig2`, and the four output signals, because their initial values are bitwise zero.

Build the model.

```
rtwbuild(model)
```

```
### Starting build procedure for model: rtwdemo_rootlevel_zero_initialization
### Successful completion of build procedure for model: rtwdemo_rootlevel_zero_initial
```

Here is the `rtwdemo_rootlevel_zero_initialization.c` optimized code in the initialization function.

```
cfile = fullfile(cgDir,'rtwdemo_rootlevel_zero_initialization_ert_rtw',...
    'rtwdemo_rootlevel_zero_initialization.c');
rtwdemodbtype(cfile, 'rtwdemo_rootlevel_zero_initialization_initialize',...
    'trailer for generated code', 1, 0);
```

```
void rtwdemo_rootlevel_zero_initialization_initialize(void)
{
  /* Registration code */

  /* external inputs */
  sig1_b = -3;
  sig2_b = -3;
}

/*
```

Close the model and the code generation report.

```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

See Also

"Remove root level I/O zero initialization" (Simulink)

Related Examples

- "Optimization Tools and Techniques" on page 53-7

- “Remove Initialization Code” on page 56-3

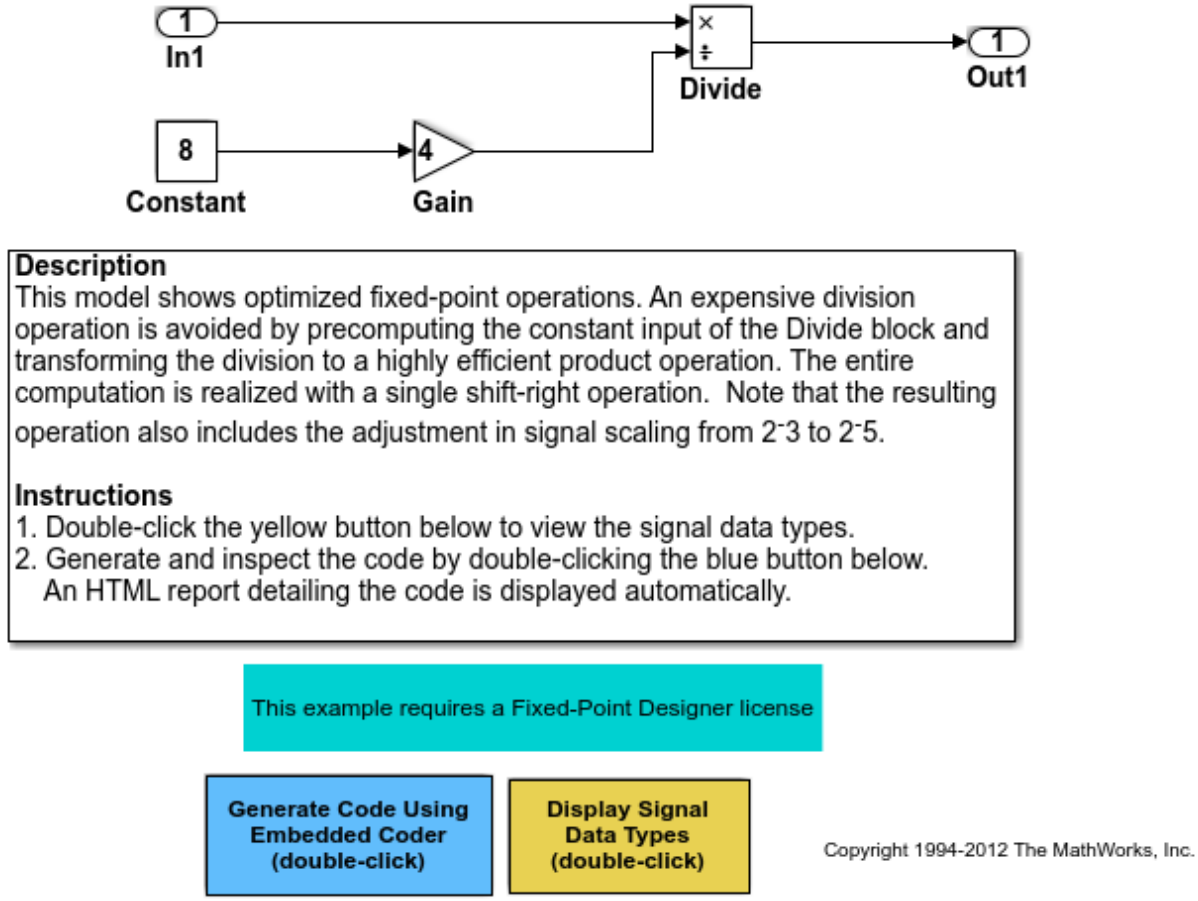
Optimize Generated Code for Fixed-Point Data Operations

This example shows how the code generater optimizes fixed-point operations by replacing expensive division operations with highly efficient product operations. This optimization improves execution speed.

Example Model

In the model rtwdemo_fixptdiv, two fixed point signals connect to a Product block. The **Number of inputs** parameter has the value `/*`.

```
model='rtwdemo_fixptdiv';
open(model);
```



Generate Code

Create a temporary folder for the build and inspection process.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
```

Build the model.

```
set_param(model,'GenCodeOnly','on');
rtwbuild(model);
```

```
### Starting build procedure for model: rtwdemo_fixptdiv
### Successful completion of code generation for model: rtwdemo_fixptdiv
```

View the generated code. Here is a portion of `rtwdemo_fixptdiv.c`.

```
cfile = fullfile(cgDir,'rtwdemo_fixptdiv_ert_rtw','rtwdemo_fixptdiv.c');
rtwdemodbtype(cfile,'/* Model step','/* Model initialize', 1, 0);
```

```
/* Model step function */
void rtwdemo_fixptdiv_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Inport: '<Root>/In1'
   *  Product: '<Root>/Divide'
   */
  rtY.Out1 = (int16_T)(rtU.In1 >> 3);
}
```

The generated code contains a highly efficient right shift operation instead of an expensive division operation. The generated code also contains the precomputed value for the constant input to the Product block.

Note that the resulting operation also includes the adjustment in signal scaling from 2^-3 to 2^-5.

Close the model and code generation report.

```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

Related Examples

- “Optimization Tools and Techniques” on page 53-7
- “Fixed Point” (Simulink)

Memory Usage in Embedded Coder

Optimize Generated Code Using Minimum and Maximum Values

To optimize the generated code for your model, you can choose an option to use input range information, also known as *design minimum and maximum*, that you specify on signals and parameters. These minimum and maximum values usually represent environmental limits, such as temperature, or mechanical and electrical limits, such as output ranges of sensors.

In the Configuration Parameters dialog box, on the **Optimization** tab, when you select the **Optimize using specified minimum and maximum values** check box, the software uses the minimum and maximum values to derive range information for downstream signals in the model. It then uses this derived range information to determine if it is possible to streamline the generated code by:

- Reducing expressions to constants
- Removing dead branches of conditional statements
- Eliminating unnecessary mathematical operations

This optimization results in:

- Reduced ROM and RAM consumption
- Improved execution speed

Configure Your Model

To make optimization more likely:

- Provide as much design minimum and maximum information as possible. Specify minimum and maximum values for signals and parameters in the model for:
  - Inport and Outport blocks
  - Block outputs
  - Block inputs, for example, for the MATLAB Function and Stateflow Chart blocks
  - `Simulink.Signal` objects
- Before generating code, test the minimum and maximum values for signals and parameters. Otherwise, optimization might result in numerical mismatch with simulation. You can simulate your model with simulation range checking enabled. If errors or warnings occur, fix these issues before generating code.

Enable Simulation Range Checking

1. In your model, select **Simulation** > **Model Configuration Parameters** to open the Configuration Parameters dialog box.
2. In the Configuration Parameters dialog box, select **Diagnostics** > **Data Validity**.
3. On the **Data Validity** pane, under **Signals**, set **Simulation range checking** to `warning` or `error`.

- Provide design minimum and maximum information upstream of blocks as close to the inputs of the blocks as possible. If you specify minimum and maximum values for a block output, these values are most likely to affect the outputs of the blocks immediately downstream.

Optimize Generated Code Using Specified Minimum and Maximum Values

This example shows how the minimum and maximum values specified on signals and parameters in a model are used to optimize the generated code.

Overview

The specified minimum and maximum values usually represent environmental limits, such as temperature, or mechanical and electrical limits, such as output ranges of sensors.

This optimization uses these values to streamline the generated code. For example, it reduces expressions to constants or removes dead branches of conditional statements.

NOTE: Make sure the minimum and maximum values that you specify are valid limits. Otherwise, this optimization might result in numerical mismatch with simulation.

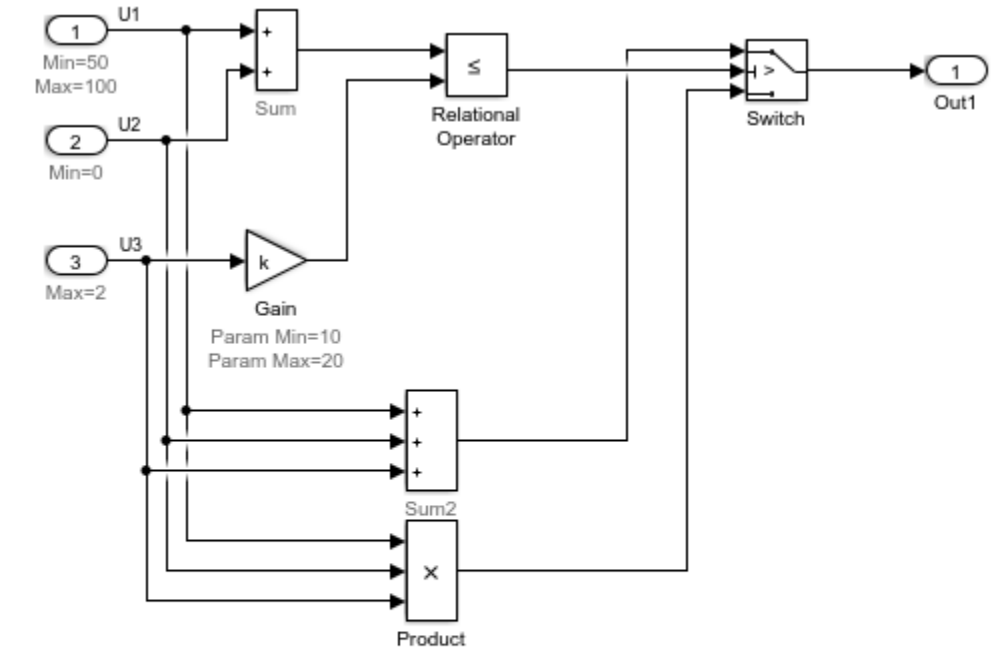
The benefits of optimizing the generated code are:

- Reducing the ROM and RAM consumption.
- Improving the execution speed.

Review Minimum and Maximum Information

Consider the model rtwdemo_minmax. In this model, there are minimum and maximum values specified on Inports and on the gain parameter of the Gain block.

```
model = 'rtwdemo_minmax';
open_system(model);
```



Generate Code Without This Optimization

First, generate code for this model without considering the min and max values.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
rtwconfiguredemo(model,'ERT')
rtwbuild(model)
```

```
### Starting build procedure for model: rtwdemo_minmax
### Successful completion of build procedure for model: rtwdemo_minmax
```

A portion of `rtwdemo_minmax.c` is listed below.

```
cfile = fullfile(cgDir,'rtwdemo_minmax_ert_rtw','rtwdemo_minmax.c');
rtwdemodbtype(cfile,'/* Model step', '/* Model initialize', 1, 0);
```

```
/* Model step function */
void rtwdemo_minmax_step(void)
{
  /* Switch: '<Root>/Switch' incorporates:
   *  Gain: '<Root>/Gain'
   *  Inport: '<Root>/U1'
   *  Inport: '<Root>/U2'
   *  Inport: '<Root>/U3'
   *  RelationalOperator: '<Root>/Relational Operator'
   *  Sum: '<Root>/Sum'
   */
  if (U1 + U2 <= k * U3) {
    /* Outport: '<Root>/Out1' incorporates:
     *  Sum: '<Root>/Sum2'
     */
    rtY.Out1 = (U1 + U2) + U3;
  } else {
    /* Outport: '<Root>/Out1' incorporates:
     *  Product: '<Root>/Product'
     */
    rtY.Out1 = U1 * U2 * U3;
  }

  /* End of Switch: '<Root>/Switch' */
}
```

Enable This Optimization

1. Open the Configuration Parameters dialog box.
2. On the **Optimization** pane, select **Optimize using the specified minimum and maximum values**.

Alternatively, you can enable this optimization by setting the command-line parameter.

```
set_param(model, 'UseSpecifiedMinMax', 'on');
```

Generate Code With This Optimization

In the model, with the specified minimum and maximum values for U1 and U2, the sum of U1 and U2 has a minimum value of 50. Considering the range of U3 and the specified minimum and maximum values for the Gain block parameter, the maximum value of the Gain block's output is 40.

The output of the Relational Operator block remains false, and the output of the Switch block remains the product of the three inputs.

Configure and build the model using Embedded Coder.

```
rtwconfiguredemo(model,'ERT')
rtwbuild(model)
```

```
### Starting build procedure for model: rtwdemo_minmax
### Successful completion of build procedure for model: rtwdemo_minmax
```

View the optimized code from `rtwdemo_minmax.c`.

```
cfile = fullfile(cgDir,'rtwdemo_minmax_ert_rtw','rtwdemo_minmax.c');
rtwdemodbtype(cfile,'/* Model step', '/* Model initialize', 1, 0);
```

```
/* Model step function */
void rtwdemo_minmax_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Inport: '<Root>/U1'
   *  Inport: '<Root>/U2'
   *  Inport: '<Root>/U3'
   *  Product: '<Root>/Product'
   *  Switch: '<Root>/Switch'
   */
  rtY.Out1 = U1 * U2 * U3;
}
```

Close the model and cleanup.

```
bdclose(model)
rtwdemoclean;
```

```
cd(currentDir)
```

Limitations

- This optimization does not take into account minimum and maximum values for:
  - Merge block inputs. To work around this issue, use a `Simulink.Signal` object on the Merge block output and specify the range on this object.
  - Bus elements.
  - Conditionally-executed subsystem (such as a triggered subsystem) block outputs that are directly connected to an Outport block.

    Outport blocks in conditionally-executed subsystems can have an initial value specified for use only when the system is not triggered. In this case, the optimization cannot use the range of the block output because the range might not cover the initial value of the block.
- If you use Polyspace software to verify code generated using this optimization, it might mark code that was previously green as orange. For example, if your model contains a division where the range of the denominator does not include zero, the generated code does not include protection against division by zero. Polyspace might mark this code orange because it does not have information about the minimum and maximum values for the inputs to the division.

  Polyspace Code Prover automatically captures some minimum and maximum values specified in the MATLAB workspace, for example, for `Simulink.Signal` and `Simulink.Parameter` objects. In this example, to provide range information to the Polyspace software, use a `Simulink.Signal` object on the input of the division and specify a range that does not include zero.

  Polyspace Code Prover stores these values in a Data Range Specification (DRS) file. However, they do not capture all minimum and maximum values in your Simulink model. To provide additional minimum and maximum information to Polyspace, you can manually define a DRS file.
- If you are using double-precision data types and the **Code Generation > Interface > Support non-finite numbers** configuration parameter is selected, this optimization does not occur.
- If your model contains multiple instances of a reusable subsystem and each instance uses input signals with different minimum and maximum values, this optimization might result in different generated code for each subsystem so code reuse does not

occur. Without this optimization, code is generated once for the subsystem and shares this code among the multiple instances of the subsystem.

- The Model Advisor DO-178C/DO-331 check **Check safety-related optimization settings** generates a warning if this option is selected. For many safety-critical applications, removing dead code automatically is unacceptable because doing so might make code untraceable. For more information about using the check to comply with DO-178C/DO-331, see Check safety-related optimization settings (Simulink Verification and Validation).

See Also

"Optimize using the specified minimum and maximum values" (Simulink)

Related Examples

- "Optimization Tools and Techniques" on page 53-7
- "Signal Ranges" (Simulink)

Flat Structures for Reusable Subsystem Parameters

This example shows how to increase the efficiency of code generated for reusable subsystems by generating a single flat parameter structure instead of a hierarchy of nested parameter structures.

By default, the code generated for reusable subsystems contains separate structures to define the parameters that each subsystem uses. If you use nested reusable subsystems, the generated code creates a hierarchy of nested parameter structures. Hierarchies of structures can reduce code efficiency due to compiler padding between word boundaries in memory.

This optimization is for only ERT-based targets. You must set the configuration parameter **Default parameter behavior** to `Inlined`.

Explore Example Model

Open the example model rtwdemo_paramstruct.

```
model = 'rtwdemo_paramstruct';
open_system(model);
```

The model contains two nested reusable subsystems. Each subsystem uses two of the parameters `A`, `B`, `C`, and `D` that are defined in the base workspace.

Generate Code with Hierarchical Parameter Structures

Create a temporary folder to contain the model build files. Generate code for the model using the default hierarchical data structure for reusable subsystems.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
rtwbuild(model)
```

In the code generation report, view the parameter structure definitions in the file `rtwdemo_paramstruct.h`.

```
cfile = fullfile(cgDir,'rtwdemo_paramstruct_ert_rtw','rtwdemo_paramstruct.h');
rtwdemodbtype(cfile,'/* Parameters for system: ''<S1>/SubsysZ'''',...
    '/* Parameters (auto storage)', 1, 0);
```

The code defines a parameter structure for each reusable subsystem and nests the structures.

Enable Optimization

Open the Configuration Parameters dialog box. On the **Optimization > Signals and Parameters** pane, select `Nonhierarchical` in the **Parameter structure** drop-down list.

Alternatively, enable the optimization at the command prompt.

```
set_param(model, 'InlinedParameterPlacement', 'NonHierarchical');
```

Generate Code with Flat Parameter Structure

Generate code for the model using a flat parameter structure for reusable subsystems.

```
rtwbuild(model)
```

In the code generation report, view the parameter structure definition in the file `rtwdemo_paramstruct.h`.

```
rtwdemodbtype(cfile,'/* Parameters (auto storage) */',...
    '/* Real-time Model Data Structure */', 1, 0);
```

The code stores all of the parameters for the reusable subsystems in a single flat structure.

Close the model and delete build files.

```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

See Also

"Parameter structure" (Simulink)

Related Examples

- "Optimization Tools and Techniques" on page 53-7
- "Optimize Generated Code By Passing Reusable Subsystem Outputs as Individual Arguments" on page 57-18
- "Reduce Global Variables in Nonreusable Subsystem Functions" on page 57-11
- "Default Data Structures in the Generated Code" on page 19-16

Reduce Global Variables in Nonreusable Subsystem Functions

In this section...

Global variables can increase memory requirements and reduce execution speed. To reduce global RAM for a nonreusable subsystem, you can generate a function interface that passes data through arguments instead of global variables. The Subsystem block parameter "Function interface" (Simulink) provides this option. To compare the outputs for the **Function interface** options, first generate a function for a subsystem with a `void-void` interface, and then generate a function with arguments.

Generate `void-void` Function

By default, when you configure a Subsystem block as a nonreusable function, it generates a `void-void` interface.

1. Open the example model `rtwdemo_roll`.
2. Right-click the subsystem `RollAngleReference`. From the list select `Block Parameter (Subsystem)`.
3. In the Block Parameter dialog box, confirm that the **Treat as atomic unit** check box is selected.
4. Click the **Code Generation** tab and set the **Function packaging** parameter to `Nonreusable function`.
5. The **Function interface** parameter is already set to `void_void`.
6. Click **Apply** and **OK**.
7. Repeat steps 2–6, for the other subsystems `HeadingMode` and `BasicRollMode`.
8. Generate code and the static code metrics report for `rtwdemo_roll`. This model is configured to generate a code generation report and to open the report automatically. For more information, see "Generate Static Code Metrics Report for Simulink Model" on page 35-38.

In the code generation report, in `rtwdemo_roll.c`, the generated code for subsystem `RollAngleReference` contains a `void-void` function definition:

```
void rtwdemo_roll_RollAngleReference(void)
```

```
{
  ...
}
```

In the static code metrics report, navigate to **Global Variables**. With the `void_void` option, the number of bytes for global variables is `59`.

2. Global Variables [hide]

Global variables defined in the generated code.

Global Variable	Size (bytes)	Reads / Writes	Reads / Writes in a Function
[+] rtwdemo_roll_U	26	14	6
[+] rtwdemo_roll_B	16	16	8
[+] rtwdemo_roll_DW	9	18	15
[+] rtwdemo_roll_M_	4	0*	0*
[+] rtwdemo_roll_Y	4	2	2
Total	**59**	**50**	

* The global variable is not directly used in any function.

Next, generate the same function with the `Allow arguments` option to compare the results.

Generate Function with Arguments

To reduce global RAM, improve ROM usage and execution speed, generate a function that allows arguments:

1. Open the Subsystem Block Parameter dialog box for `RollAngleReference`.
2. Click the **Code Generation** tab. Set the **Function interface** parameter to `Allow arguments`.
3. Click **Apply** and **OK**.
4. Repeat steps 2 and 3, for the other subsystems `HeadingMode` and `BasicRollMode`.
5. Generate code and the static code metrics report for `rtwdemo_roll`.

In the code generation report, in `rtwdemo_roll.c`, the generated code for subsystem `RollAngleReference` now has arguments:

```
real32_T rtwdemo_roll_RollAngleReference(real32_T rtu_Turn_Knob,...
                                         boolean_T rtu_AP_Eng,...
                                         real32_T rtu_Phi)
 {
 ...
 }
```

In the static code metrics report, navigate to **Global Variables**. With the `Allow arguments` option set, the total number of bytes for global variables is now `47` bytes.

2. Global Variables [hide]

Global variables defined in the generated code.

Global Variable	Size (bytes)	Reads / Writes	Reads / Writes in a Function
[+] rtwdemo_roll_U	26	11	8
[+] rtwdemo_roll_DW	9	18	15
[+] rtwdemo_roll_B	4	2	1
[+] rtwdemo_roll_M_	4	0*	0*
[+] rtwdemo_roll_Y	4	2	2
Total	**47**	**33**	

* The global variable is not directly used in any function.

See Also

"Function interface" (Simulink)

Related Examples

- "Optimization Tools and Techniques" on page 53-7
- "Generate Subsystem Code as Separate Function and Files" on page 3-10
- "Flat Structures for Reusable Subsystem Parameters" on page 57-9
- "Optimize Generated Code By Passing Reusable Subsystem Outputs as Individual Arguments" on page 57-18

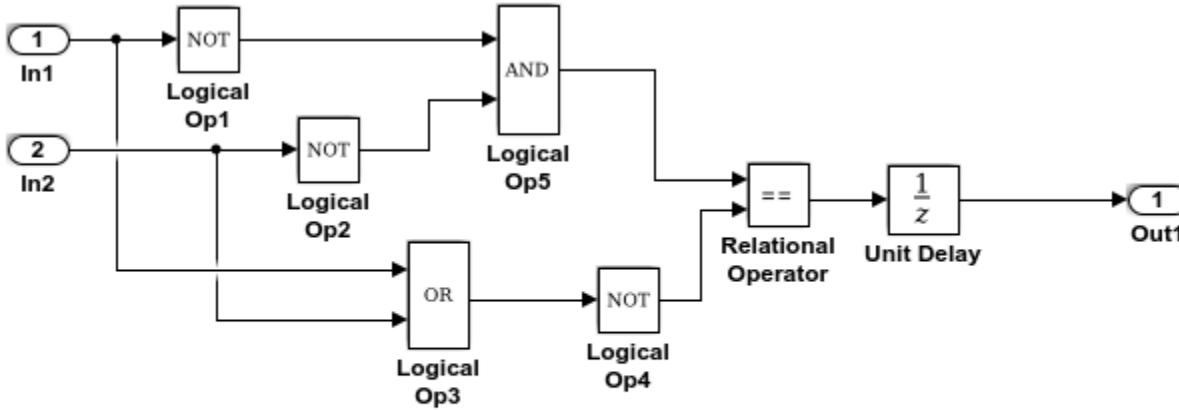
Optimize Generated Code By Packing Boolean Data Into Bitfields

This example shows how to optimize the generated code by packing Boolean data into bitfields. When you select the model configuration parameter **Pack Boolean data into bitfields**, Embedded Coder® packs the Boolean signals into 1-bit bitfields, reducing RAM consumption. By default, the optimization is enabled. This optimization reduces the RAM consumption. Be aware that this optimization can potentially increase code size and execution speed.

Example Model

Consider the model rtwdemo_pack_boolean.

```
model = 'rtwdemo_pack_boolean';
open_system(model);
```



Disable Optimization

1. Open the Configuration Parameters dialog box.
2. On the **Optimization > Signals and Parameters** pane, clear **Pack Boolean data into bitfields**.

Alternatively, you can use the command-line API to disable the optimization:

```
set_param(model,'BooleansAsBitfields','off');
```

Create a temporary folder (in your system temporary folder) for the build and inspection process.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
```

Generate Code Without Optimization

Build the model using Embedded Coder®.

```
rtwbuild(model)
```

```
### Starting build procedure for model: rtwdemo_pack_boolean
### Successful completion of build procedure for model: rtwdemo_pack_boolean
```

View the generated code without the optimization. These lines of code are in `rtwdemo_pack_boolean.h`.

```
hfile = fullfile(cgDir,'rtwdemo_pack_boolean_ert_rtw','rtwdemo_pack_boolean.h');
rtwdemodbtype(hfile,'/* Block signals and states','/* External inputs',1,0);
```

```
/* Block signals and states (auto storage) for system '<Root>' */
typedef struct {
  boolean_T LogicalOp1;                /* '<Root>/Logical Op1' */
  boolean_T LogicalOp2;                /* '<Root>/Logical Op2' */
  boolean_T LogicalOp5;                /* '<Root>/Logical Op5' */
  boolean_T LogicalOp3;                /* '<Root>/Logical Op3' */
  boolean_T LogicalOp4;                /* '<Root>/Logical Op4' */
  boolean_T RelationalOperator;        /* '<Root>/Relational Operator' */
  boolean_T UnitDelay_DSTATE;          /* '<Root>/Unit Delay' */
} DW;
```

Enable Optimization

1. Open the Configuration Parameters dialog box.
2. On the **Optimization > Signals and Parameters** pane, select **Pack Boolean data into bitfields**.

Alternatively, you can use the command-line API to enable the optimization:

```
set_param(model,'BooleansAsBitfields','on');
```

Generate Code with Optimization

Build the model using Embedded Coder®.

```
rtwbuild(model)
```

```
### Starting build procedure for model: rtwdemo_pack_boolean
### Successful completion of build procedure for model: rtwdemo_pack_boolean
```

View the generated code with the optimization. These lines of code are in `rtwdemo_pack_boolean.h`.

```
hfile = fullfile(cgDir,'rtwdemo_pack_boolean_ert_rtw','rtwdemo_pack_boolean.h');
rtwdemodbtype(hfile,'/* Block signals and states','/* External inputs',1,0);
```

```
/* Block signals and states (auto storage) for system '<Root>' */
typedef struct {
  struct {
    uint_T LogicalOp1:1;               /* '<Root>/Logical Op1' */
    uint_T LogicalOp2:1;               /* '<Root>/Logical Op2' */
    uint_T LogicalOp5:1;               /* '<Root>/Logical Op5' */
    uint_T LogicalOp3:1;               /* '<Root>/Logical Op3' */
    uint_T LogicalOp4:1;               /* '<Root>/Logical Op4' */
    uint_T RelationalOperator:1;       /* '<Root>/Relational Operator' */
    uint_T UnitDelay_DSTATE:1;         /* '<Root>/Unit Delay' */
  } bitsForTID0;
} DW;
```

Selecting **Pack Boolean data into bitfields** enables model configuration parameter **Bitfield declarator type specifier**. To optimize your code further, select `uchar_t`. However, the optimization benefit of the **Bitfield declarator type specifier** setting depends on your choice of target.

Close the model and code generation report.

```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

See Also

"Pack Boolean data into bitfields" (Simulink)

Related Examples

- "Optimization Tools and Techniques" on page 53-7
- "Optimize Generated Code Using Boolean Data for Logical Signals" on page 53-87
- "Replace `boolean` with Specific Integer Data Type" on page 56-14
- "Data Types Supported by Simulink" (Simulink)

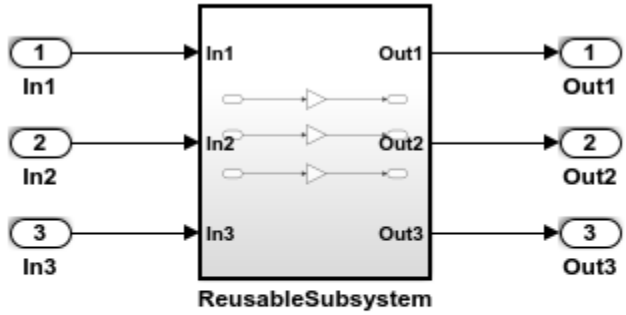
Optimize Generated Code By Passing Reusable Subsystem Outputs as Individual Arguments

This example shows how passing reusable subsystem outputs as individual arguments instead of as a pointer to a structure stored in global memory optimizes the generated code. This optimization conserves RAM consumption and increases code execution speed by reducing global memory usage and eliminating data copies from local variables back to global block I/O structures.

Example Model

Consider the model rtwdemo_reusable_sys_outputs. In this model, the reusable subsystem outputs feed the root outputs of the model.

```
model = 'rtwdemo_reusable_sys_outputs';
open_system(model);
```



Generate Code Without This Optimization

Generate code for this model while passing subsystem outputs as a structure reference. Create a temporary folder for the build and inspection process.

```
currentDir = pwd;
```

```
[~,cgDir] = rtwdemodir();
```

Build the model.

```
rtwbuild(model)
```

```
### Starting build procedure for model: rtwdemo_reusable_sys_outputs
### Successful completion of build procedure for model: rtwdemo_reusable_sys_outputs
```

The code snippet shows portions of `rtwdemo_reusable_sys_outputs.c`. Notice the global block I/O structure and in the model step function a data copy from this structure.

```
cfile = fullfile(cgDir,'rtwdemo_reusable_sys_outputs_ert_rtw',...
'rtwdemo_reusable_sys_outputs.c');
rtwdemodbtype(cfile,'/* Output and update for atomic system',...
'/* Model initialize', 1, 0);
```

```
/* Output and update for atomic system: '<Root>/ReusableSubsystem' */
void ReusableSubsystem(real_T rtu_In1, real_T rtu_In2, real_T rtu_In3,
  DW_ReusableSubsystem *localDW)
{
  /* Gain: '<S1>/Gain' */
  localDW->Gain = 5.0 * rtu_In1;

  /* Gain: '<S1>/Gain1' */
  localDW->Gain1 = 6.0 * rtu_In2;

  /* Gain: '<S1>/Gain2' */
  localDW->Gain2 = 7.0 * rtu_In3;
}

/* Model step function */
void rtwdemo_reusable_sys_outputs_step(void)
{
  /* Outputs for Atomic SubSystem: '<Root>/ReusableSubsystem' */

  /* Inport: '<Root>/In1' incorporates:
   *  Inport: '<Root>/In2'
   *  Inport: '<Root>/In3'
   */
  ReusableSubsystem(rtU.In1, rtU.In2, rtU.In3, &rtDW.ReusableSubsystem_d);

  /* End of Outputs for SubSystem: '<Root>/ReusableSubsystem' */
```

```
  /* Outport: '<Root>/Out1' */
  rtY.Out1 = rtDW.ReusableSubsystem_d.Gain;

  /* Outport: '<Root>/Out2' */
  rtY.Out2 = rtDW.ReusableSubsystem_d.Gain1;

  /* Outport: '<Root>/Out3' */
  rtY.Out3 = rtDW.ReusableSubsystem_d.Gain2;
}
```

Enable This Optimization

1. Open the Configuration Parameters dialog box.
2. On the **Optimization > Signals and Parameters** pane, set **Pass reusable subsystem outputs as** to `Individual arguments`.

Alternatively, you can use the command-line API to enable the optimization:

```
set_param(model, 'PassReuseOutputArgsAs', 'Individual arguments');
```

Generate Code With This Optimization

With this optimization, the `ReusableSubsystem` function has three output arguments, which are direct references to the external outputs. The `rtDW` global structure no longer exists, and the data copies from this structure to the `rtY` (external outputs) structure are not in the generated code.

Build the model.

```
rtwbuild(model)
```

```
### Starting build procedure for model: rtwdemo_reusable_sys_outputs
### Successful completion of build procedure for model: rtwdemo_reusable_sys_outputs
```

The code snippet below is a portion of `rtwdemo_reusable_sys_outputs.c`. Observe the optimized code.

```
rtwdemodbtype(cfile,'/* Output and update for atomic system',...
'/* Model initialize', 1, 0);
```

```
/* Output and update for atomic system: '<Root>/ReusableSubsystem' */
void ReusableSubsystem(real_T rtu_In1, real_T rtu_In2, real_T rtu_In3, real_T
  *rty_Out1, real_T *rty_Out2, real_T *rty_Out3)
{
```

```
  /* Gain: '<S1>/Gain' */
  *rty_Out1 = 5.0 * rtu_In1;

  /* Gain: '<S1>/Gain1' */
  *rty_Out2 = 6.0 * rtu_In2;

  /* Gain: '<S1>/Gain2' */
  *rty_Out3 = 7.0 * rtu_In3;
}

/* Model step function */
void rtwdemo_reusable_sys_outputs_step(void)
{
  /* Outputs for Atomic SubSystem: '<Root>/ReusableSubsystem' */

  /* Inport: '<Root>/In1' incorporates:
   *  Inport: '<Root>/In2'
   *  Inport: '<Root>/In3'
   *  Outport: '<Root>/Out1'
   *  Outport: '<Root>/Out2'
   *  Outport: '<Root>/Out3'
   */
  ReusableSubsystem(rtU.In1, rtU.In2, rtU.In3, &rtY.Out1, &rtY.Out2, &rtY.Out3);

  /* End of Outputs for SubSystem: '<Root>/ReusableSubsystem' */
}
```

Close the model and cleanup.

```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

See Also

"Pass reusable subsystem outputs as" (Simulink)

Related Examples

- "Optimization Tools and Techniques" on page 53-7
- "Flat Structures for Reusable Subsystem Parameters" on page 57-9
- "Virtualized Output Ports Optimization" on page 55-17
- "Reduce Global Variables in Nonreusable Subsystem Functions" on page 57-11

- “Default Data Structures in the Generated Code” on page 19-16

Convert Data Copies to Pointer Assignments

The code generator optimizes generated code for vector signal assignments by trying to replace `for` loop controlled element assignments and `memcpy` function calls with pointer assignments. Pointer assignments avoid expensive data copies. Therefore, they use less stack space and offer faster execution speed than `for` loop controlled element assignments and `memcpy` function calls. If you assign large data sets to vector signals, this optimization can result in significant improvements to code efficiency.

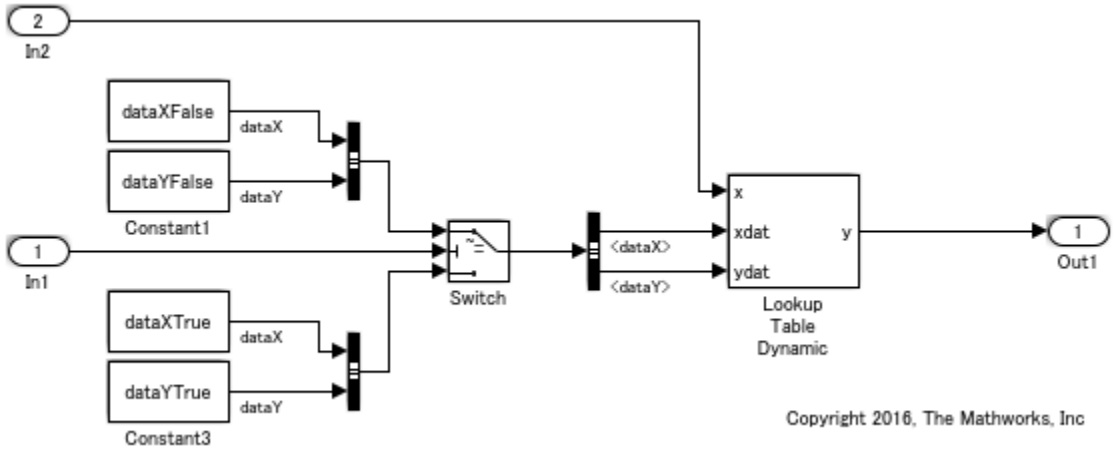
Configure Model to Optimize Generated Code for Vector Signal Assignments

To apply this optimization:

1. Verify that your target supports the `memcpy` function.
2. Determine whether your model uses vector signal assignments (such as `Y=expression`) to move large amounts of data. For example, your model could use a Selector block to select input elements from a vector, matrix, or multidimension signal.
3. On the **Optimization > Signals and Parameters** pane, the **Use memcpy for vector assignment parameter**, which is on by default, enables the associated **Memcpy threshold (bytes)** parameter.
4. Examine the setting of **Memcpy threshold (bytes)**. By default, it specifies 64 bytes as the minimum array size for which `memcpy` function calls or pointer assignments can replace `for` loops in the generated code. Based on the array sizes in your application's vector signal assignments, and target environment considerations on the threshold selection, accept the default value or specify another array size.

Example Model

Consider the following model named rtwdemo_pointer_conversion. This model uses a Switch block to assign data to a vector signal. This signal then feeds into a Bus Selector block.



Generate Code without Optimization

1. In the Configuration Parameters dialog box, on the **All Parameters** tab, clear the **Use memcpy for vector assignment** parameter.
2. Create a temporary folder for the build and inspection process.
3. Press **Ctrl+B** to generate code.

```
### Starting build procedure for model: rtwdemo_pointer_conversion
### Successful completion of build procedure for model: rtwdemo_pointer_conversion
```

View the generated code without the optimization. Here is a portion of `rtwdemo_pointer_conversion.c` .

```
/* Model step function */
void rtwdemo_pointer_conversion_step(void)
{
  int16_T rtb_dataX[100];
  int16_T rtb_dataY[100];
  int32_T i;

  /* Switch: '<Root>/Switch' incorporates:
   *  Constant: '<Root>/Constant'
   *  Constant: '<Root>/Constant1'
   *  Constant: '<Root>/Constant2'
   *  Constant: '<Root>/Constant3'
```

```
   *  Inport: '<Root>/In1'
   */
  for (i = 0; i < 100; i++) {
    if (rtU.In1) {
      rtb_dataX[i] = rtCP_Constant_Value[i];
      rtb_dataY[i] = rtCP_Constant1_Value[i];
    } else {
      rtb_dataX[i] = rtCP_Constant2_Value[i];
      rtb_dataY[i] = rtCP_Constant3_Value[i];
    }
  }

  /* End of Switch: '<Root>/Switch' */

  /* S-Function (sfix_look1_dyn): '<Root>/Lookup Table Dynamic' incorporates:
   *  Inport: '<Root>/In2'
   *  Outport: '<Root>/Out1'
   */
  /* Dynamic Look-Up Table Block: '<Root>/Lookup Table Dynamic'
   * Input0  Data Type:  Integer        S16
   * Input1  Data Type:  Integer        S16
   * Input2  Data Type:  Integer        S16
   * Output0 Data Type:  Integer        S16
   * Lookup Method: Linear_Endpoint
   *
   */
  LookUp_S16_S16( &(rtY.Out1), &rtb_dataY[0], rtU.In2, &rtb_dataX[0], 99U);
}
```

Without the optimization, the generated code contains `for` loop controlled element assignments.

Enable Optimization and Generate Code

1. In the Configuration Parameter dialog box, on the **All Parameters** tab, select the **Use memcpy for vector assignment** parameter.
2. Generate code.

```
### Starting build procedure for model: rtwdemo_pointer_conversion
### Successful completion of build procedure for model: rtwdemo_pointer_conversion
```

View the generated code without the optimization. Here is a portion of `rtwdemo_pointer_conversion.c`.

```
/* Model step function */
void rtwdemo_pointer_conversion_step(void)
{
  const int16_T *rtb_dataX_0;
  const int16_T *rtb_dataY_0;

  /* Switch: '<Root>/Switch' incorporates:
   *  Inport: '<Root>/In1'
   */
  if (rtU.In1) {
    /* Switch: '<Root>/Switch' incorporates:
     *  Constant: '<Root>/Constant'
     *  Constant: '<Root>/Constant1'
     */
    rtb_dataX_0 = &rtCP_Constant_Value[0];
    rtb_dataY_0 = &rtCP_Constant1_Value[0];
  } else {
    /* Switch: '<Root>/Switch' incorporates:
     *  Constant: '<Root>/Constant2'
     *  Constant: '<Root>/Constant3'
     */
    rtb_dataX_0 = &rtCP_Constant2_Value[0];
    rtb_dataY_0 = &rtCP_Constant3_Value[0];
  }

  /* S-Function (sfix_look1_dyn): '<Root>/Lookup Table Dynamic' incorporates:
   *  Inport: '<Root>/In2'
   *  Outport: '<Root>/Out1'
   */
  /* Dynamic Look-Up Table Block: '<Root>/Lookup Table Dynamic'
   * Input0  Data Type:  Integer        S16
   * Input1  Data Type:  Integer        S16
   * Input2  Data Type:  Integer        S16
   * Output0 Data Type:  Integer        S16
   * Lookup Method: Linear_Endpoint
   *
   */
  LookUp_S16_S16( &(rtY.Out1), &rtb_dataY_0[0], rtU.In2, &rtb_dataX_0[0], 99U);
}
```

Because the setting of the **Memcpy threshold (bytes)** parameter is below the array sizes in the generated code, the optimized code contains pointer assignments for the vector signal assignments.

See Also

"Use memcpy for vector assignment" (Simulink) | "Memcpy threshold (bytes)" (Simulink)

Related Examples

- "Optimization Tools and Techniques" on page 53-7
- "Use `memcpy` Function to Optimize Generated Code for Vector Assignments" on page 53-52
- "Vector Operation Optimization" on page 53-97

Remove Reset and Disable Functions from the Generated Code

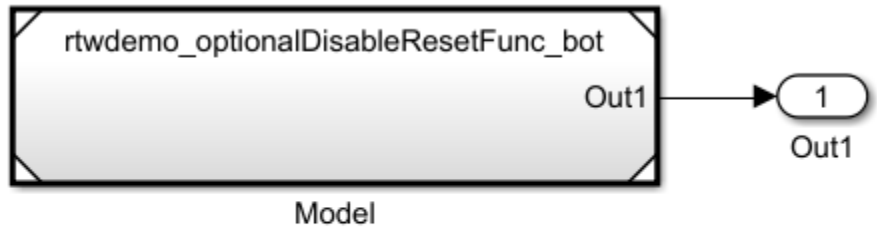
In this section...

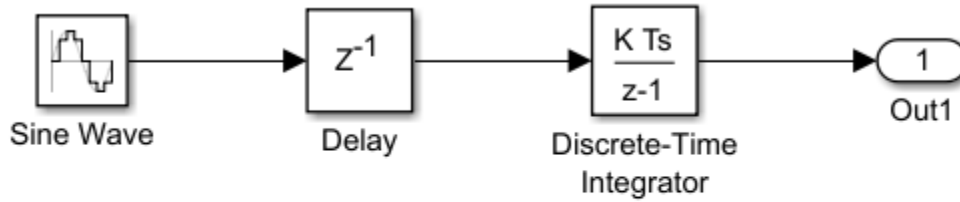
This example shows how the code generator removes unreachable (dead code) instances of the reset and disable functions from the generated code for ERT-based systems that include model referencing hierarchies. Optimizing the generated code to remove unreachable code is a requirement for safety-critical systems. This optimization also improves execution speed and reduces ROM consumption.

If a model contains blocks with states, the generated code contains reset and disable functions. If the model is not part of a conditionally executed system, such as an enabled subsystem, the code generator can remove the disable function because the generated code does not call it. If the model is not part of a conditionally executed system that can reset states when a control input enables it, the code generator can remove the reset function because the generated code does not call it.

Example Model

A referenced model, `rtwdemo_optionalDisableResetFunc_bot`, is in `rtwdemo_optionalDisableResetFunc_top`. The referenced model contains two blocks with states, a Delay block and a Discrete-Time Integrator block.





Generate Code

1. Open the models. In the Command Window, type `rtwdemo_optionalDisableResetFunc_bot` and `rtwdemo_optionalDisableResetFunc_top`.
2. In your system temporary folder, create a temporary folder for the build and inspection process.
3. Build the model.
4. Open the `rtwdemo_optionalDisableResetFunc_top.c` and `rtwdemo_optionalDisableResetFunc_bot.c` files.

The `rtwdemo_optionalDisableResetFunc_bot.c` file contains these reset and disable functions.

```
/* System reset for referenced model: 'rtwdemo_optionalDisableResetFunc_bot' */
void rtwdemo_optionalDisableResetFunc_bot_Reset
  (DW_rtwdemo_optionalDisableResetFunc_bot_f_T *localDW)
{
  /* InitializeConditions for Delay: '<Root>/Delay' */
  localDW->Delay_DSTATE = 0.0;

  /* InitializeConditions for DiscreteIntegrator: '<Root>/Discrete-Time Integrator' */
  localDW->DiscreteTimeIntegrator_DSTATE = 3.0;
}

/* Disable for referenced model: 'rtwdemo_optionalDisableResetFunc_bot' */
void rtwdemo_optionalDisableResetFunc_bot_Disable(real_T *rty_Out1,
  DW_rtwdemo_optionalDisableResetFunc_bot_f_T *localDW)
{
  /* Disable for DiscreteIntegrator: '<Root>/Discrete-Time Integrator' */
  localDW->DiscreteTimeIntegrator_DSTATE = *rty_Out1;
}
```

The `rtwdemo_optionalDisableResetFunc_top_step` function does not call the `rtwdemo_optionalDisableResetFunc_bot_Disable` function because the model is not part of a conditionally executed system. The `rtwdemo_optionalDisableResetFunc_top_step` function does not call the `rtwdemo_optionalDisableResetFunc_bot_Reset` function because the model is not part of a conditionally executed system that can reset states when a control input enables it.

Enable Optimization

1. Open the Model Configuration Parameters dialog box for `rtwdemo_optionalDisableResetFunc_bot`.
2. On the **All Parameters** tab, select **Remove Disable Function** and **Remove Reset Function**.

Open the `rtwdemo_optionalDisableResetFunc_bot.c` file. The code does not contain the `rtwdemo_optionalDisableResetFunc_bot_Reset` function or the `rtwdemo_optionalDisableResetFunc_bot_Disable` function.

See Also

"Remove reset function" | "Remove disable function"

Related Examples

- "Optimization Tools and Techniques" on page 53-7

Code Execution Profiling in Embedded Coder

Code Execution Profiling with SIL and PIL

In this section...

You can configure a software-in-the-loop (SIL) or processor-in-the-loop (PIL) on page 64-2 simulation to produce execution-time metrics for tasks and functions in your generated code. The software calculates execution times from data that is obtained through code instrumentation added to the SIL or PIL application or the generated code under test. You can use the execution-time metrics to determine whether the generated code meets the requirements for real-time deployment on your target hardware.

For example, you can perform the following analysis:

1. Identify tasks that require the most time. Tasks are main entry points into the generated code. For example, the step function for a sample rate or the *`model`*`_initialize` function.
2. In these tasks, investigate code sections that require the most time.
3. Identify variations in execution time over time steps.

If you are trying to reduce execution times, the analysis results help you to focus on the most critical code sections. To observe performance changes for an updated model, rerun the SIL or PIL simulation and compare the new metrics against previous metrics.

Note: Execution-time measurements depend greatly on the hardware that you use. For reliable results, collect execution-time metrics using hardware on which you plan to deploy the generated code, that is, run PIL simulations that execute code on your target hardware. SIL simulations, which execute code on your host computer, might not produce representative metrics.

When the SIL or PIL simulation is complete, you can:

- View execution-time metrics through a display window or report.
- Use the Simulation Data Inspector to view and compare the variation of execution times over a simulation.
- Analyze the measurements within the MATLAB environment.

Configure Code Execution Profiling

To configure code execution profiling for a SIL or PIL simulation:

1. In your top model, open the Configuration Parameters dialog box, and select the **Code Generation** > **Verification** pane.
2. Select the **Measure task execution time** check box.
3. If you also want function execution times, select the **Measure function execution times** check box.
4. In the **Workspace variable** field, specify a name. When you run the simulation, the software generates a variable with this name in the MATLAB base workspace. The variable contains the execution-time measurements, and is an object of type `coder.profile.ExecutionTime`.

   If you select the **Data Import/Export** > **Single simulation output** check box, the software creates the variable in the `Simulink.SimulationOutput` object that you specify.
5. From the **Save options** drop-down list, select one of the following:

   - `Summary data only` — If you want to generate only a report and reduce memory usage, for example, during a long simulation.
   - `All data` — Allows you to generate a report and store execution-time profiles in the base workspace. After the simulation, you can use methods from the `coder.profile.ExecutionTime` and `coder.profile.ExecutionTimeSection` classes to retrieve execution-time measurements for every call to each profiled section of code that occurs during the simulation.
6. Click **OK**.

For a PIL simulation, you must configure a hardware-specific timer. When you set up the connectivity configuration for your target, create a timer object. This action is not required for a SIL simulation.

If you select `All data` from the **Save options** drop-down list, the metrics display window and generated report display Simulation Data Inspector icons  . When you click one of these icons, the software imports simulation results into the Simulation Data Inspector. You can then plot execution times and manage and compare plots from various simulations.

Profiling for Atomic Subsystems and Model Reference Hierarchies

To generate execution-time metrics for tasks only, on the **Code Generation** > **Verification** pane of the Configuration Parameters dialog box, select the **Measure task execution time** check box and clear the **Measure function execution times** check box.

To generate function execution data for atomic subsystems in the top model, on the **Code Generation** > **Verification** pane, select the **Measure task execution time** and **Measure function execution times** check boxes.

The generation of function execution data requires the insertion of measurement probes into the generated code. The software inserts measurement probes for an atomic subsystem only if you set the **Function packaging** field (on the **Code Generation** tab of the Function Block Parameters dialog box) to either `Nonreusable function` or `Reusable function`. If the field is set to `Auto`, then the insertion of probes depends on the packaging choice that results from the `Auto` setting. If the field is set to `Inline`, the software does not insert probes.

Note: In the generated code, the software wraps each function call with measurement probes except when:

- The call site cannot be wrapped because of expression folding (see "Minimize Computations and Storage for Intermediate Results at Block Outputs" (Simulink Coder)).
- The call site is located in the shared utility code (see "Sharing Utility Code" (Simulink Coder)).

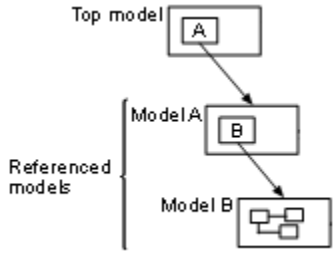
You might not want to generate profiles for specific subsystems. To disable code execution profiling for a subsystem in the top model:

1. Right-click the subsystem.
2. From the context menu, select **Properties**.
3. In the Block Properties dialog box, select the **General** tab.
4. In the **Tag** field, enter `DoNotProfile`.
5. Click **OK**.

To generate function execution data for model reference hierarchies:

1. In the top model, open the Configuration Parameters dialog box, and select the **Code Generation > Verification** pane.
2. Select the **Measure task execution time** check box.
3. For each Model block that you want to profile, select **Measure function execution times** only at the reference level for which you require function execution data.

For example, consider a top model that has Model block A, which in turn contains Model block B.



If you want to generate execution times for functions from model B, select **Measure task execution time** for the top model and **Measure function execution times** for model B.

Note: By default, the Model block parameter **Code interface** is set to `Model reference`. If this block parameter is set to `Top model`, the configuration parameter **Measure task execution time** for the top model and the referenced model must be the same. Otherwise, the software produces an error.

If your top model has a PIL block, the execution profiling settings that apply to the PIL block are the settings from the original model that you used to create the PIL block. See “Simulation with Blocks From Subsystems” on page 64-18. The execution profiling settings of your top model do not apply to the PIL block.

See Also

“Save options” | “Measure function execution times” | “Workspace variable” | “Measure task execution time”

Related Examples

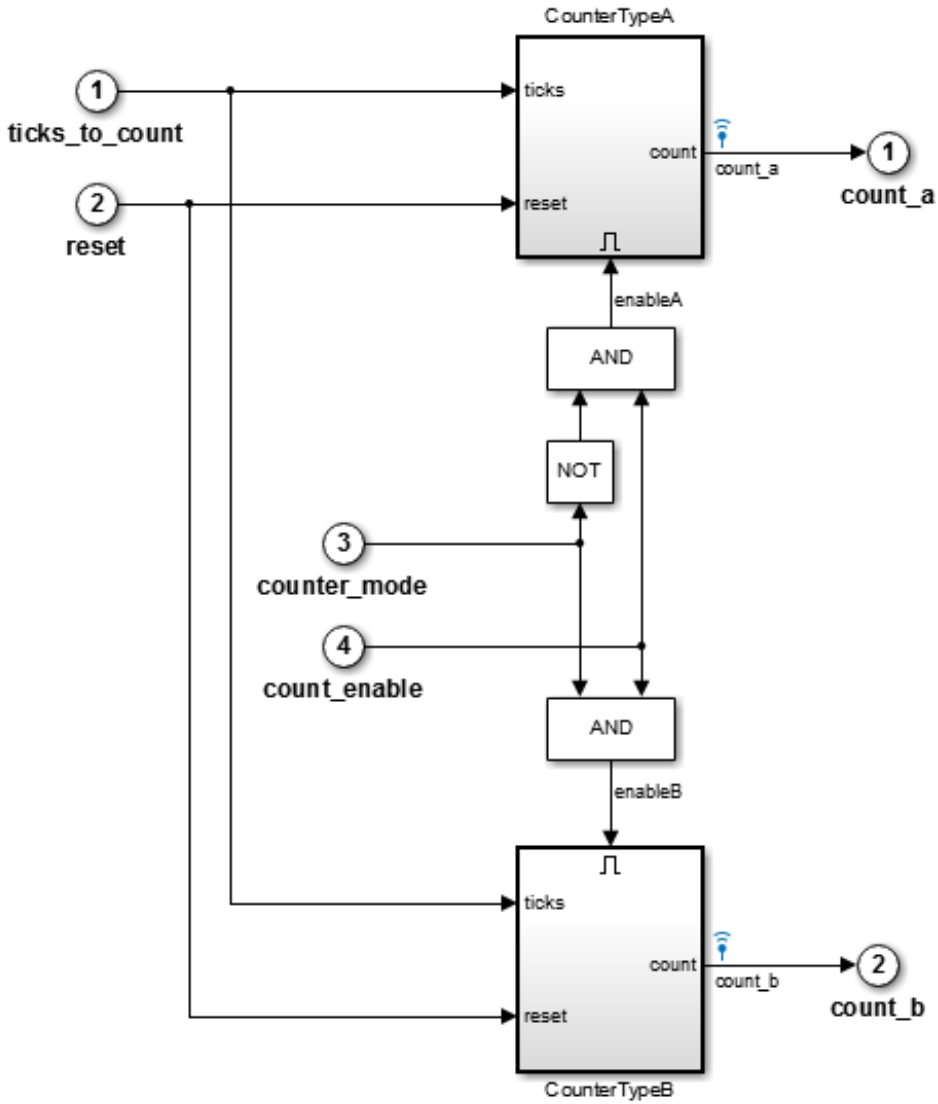
- "Configure and Run SIL Simulation" on page 64-15
- "View and Compare Code Execution Times" on page 58-7
- "Analyze Code Execution Data" on page 58-18
- "Specify Hardware Timer" on page 64-52
- "View and Analyze Simulation Results" (Simulink)

View and Compare Code Execution Times

During a software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation, you can use the Simulation Data Inspector to observe streamed execution times. At the end of the simulation, you can:

- View execution-time metrics for a profiled model component.
- Open a report of execution-time metrics for all profiled components.
- Use the Simulation Data Inspector to plot and compare execution times from various simulations.

Consider the model `rtwdemo_sil_topmodel`, which has two subsystems `CounterTypeA` and `CounterTypeB`.


CounterTypeA
1
ticks_to_count
2
reset
ticks
reset
count
count_a
1
count_a
enableA
AND
NOT
3
counter_mode
4
count_enable
AND
enableB
ticks
reset
count
count_b
2
count_b
CounterTypeB

To measure code execution times for the subsystems, on the **Configuration Parameters > Code Generation > Verification** pane:

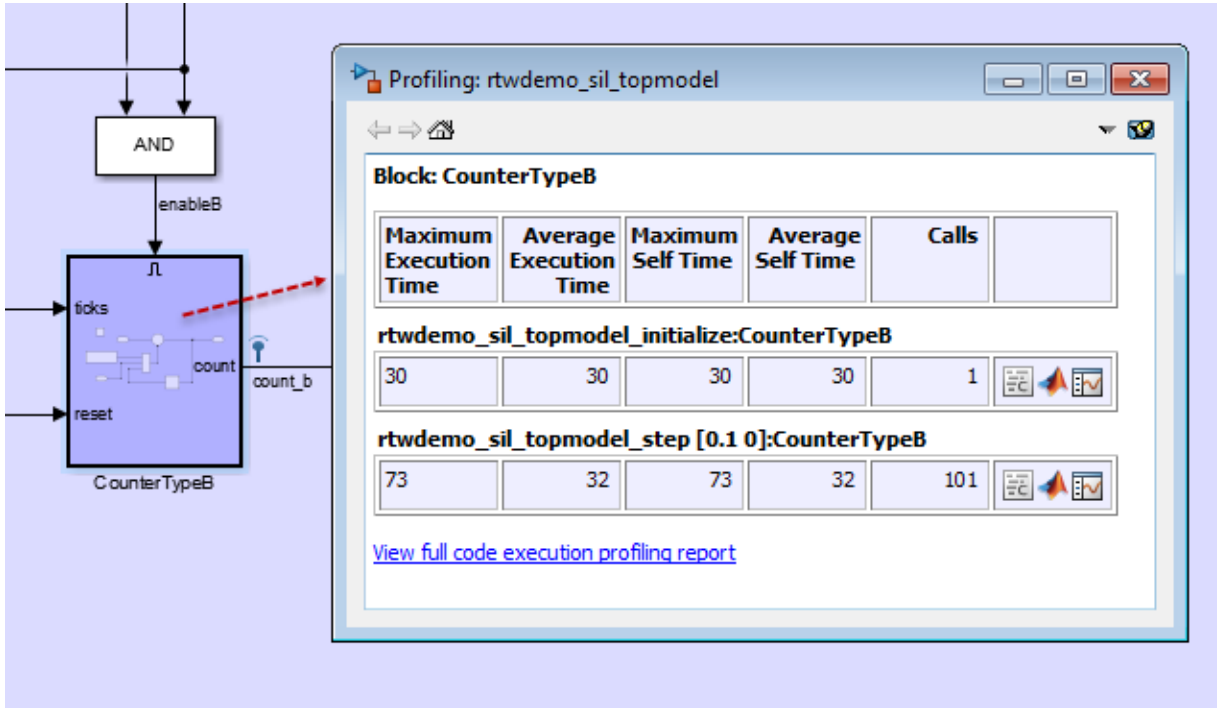
1. Select the following check boxes:
   - **Measure task execution time** — Provides execution-time metrics for the task generated from the top model `rtwdemo_sil_topmodel`.
   - **Measure function execution times** — Provides execution-time metrics for the functions generated from the subsystems `CounterTypeA` and `CounterTypeB`.
2. Specify a **Workspace variable**, for example, `executionProfile`.
3. From the **Save options** drop-down list, select `All data`.

The simulation generates the variable `executionProfile` in the MATLAB base workspace.

Note: If you select the **Data Import/Export > Single simulation output** check box, the simulation creates the variable in your specified `Simulink.SimulationOutput` object.

To view streamed execution times during the simulation, open the Simulation Data Inspector. On the Simulink Editor toolbar, click the Simulation Data Inspector button.

When the simulation is complete, the profiled model components are colored blue. To view execution-time metrics for a profiled component, click the component. For example, subsystem `CounterTypeB`.



The display window also has links to:

- The complete profiling report, which provides execution-time metrics for all profiled code sections.
- The profiled code section in the code generation report.
- The Simulation Data Inspector, which allows you to plot and compare execution-time measurements for the profiled code section.

For top-model SIL or PIL simulations, the Simulink Editor background is also colored blue. When you click the background, the display window shows execution-time metrics for top-model tasks.

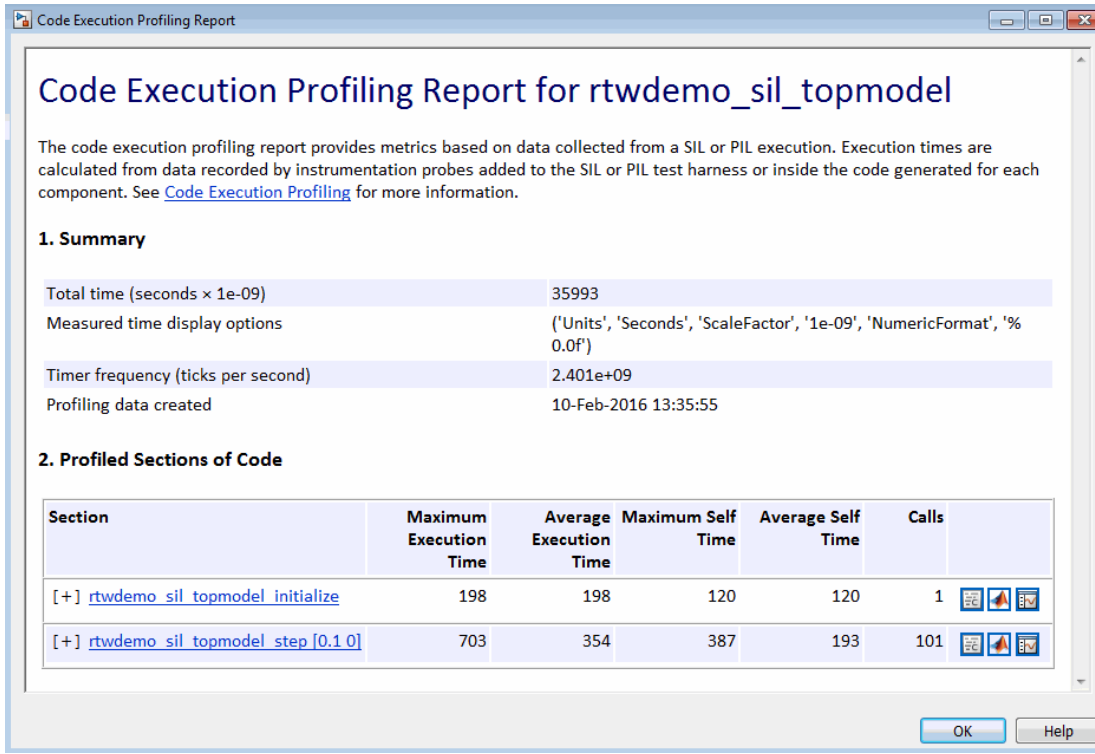
If you close the model or display window, you can reopen the colored model and display window with this line command:

```
>> annotate(executionProfile)
```

Code Execution Profiling Report

At the end of the simulation, you can open this report through the metrics display window or with this line command:

```
>> report(executionProfile)
```



Part 1 provides a summary. Part 2 contains information about profiled code sections.

Expand and collapse profiled sections in Part 2 by clicking [+] and [–] respectively. This graphic shows fully expanded sections.

2. Profiled Sections of Code

Section	Maximum Execution Time	Average Execution Time	Maximum Self Time	Average Self Time	Calls	
[−] rtwdemo_sil_topmodel_initialize	198	198	120	120	1	
CounterTypeA	52	52	52	52	1	
CounterTypeB	27	27	27	27	1	
[−] rtwdemo_sil_topmodel_step [0.1 0]	703	354	387	193	101	
CounterTypeA	257	116	257	116	101	
CounterTypeB	87	45	87	45	101	

The report contains time measurements for:

- The model initialization function `rtwdemo_sil_topmodel_initialize`.
- A task represented by the step function `rtwdemo_sil_topmodel_step [0.1 0]`.
- Functions generated from the subsystems `CounterTypeA` and `CounterTypeB`.

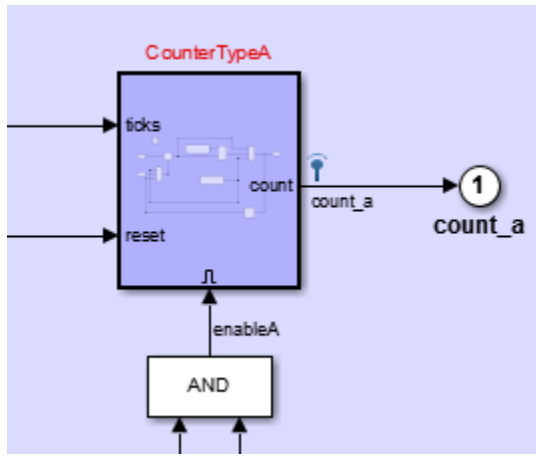
You can go to a profiled code section in the Generated Code view of the Code Generation Report. In the Code Execution Profiling Report, on a code section row, click the icon  . For example, if you click the icon for the `rtwdemo_sil_topmodel_initialize` task, you see the measurement probes around the call site in the SIL application.

```
66 XIL_INTERFACE_ERROR_CODE xilInitialize(uint32_T xilFcnId)
67 {
68   XIL_INTERFACE_ERROR_CODE errorCode = XIL_INTERFACE_SUCCESS;
69
70   /* initialize output storage owned by In-the-Loop */
71   /* Single In-the-Loop Component */
72   if (xilFcnId == 0) {
73     taskTimeStart_51c545e6ce8b10bf(1U);
74     rtwdemo_sil_topmodel_initialize();
75     taskTimeEnd_rt_38248ea98502ec29(1U);
76   } else {
77     errorCode = XIL_INTERFACE_UNKNOWN_FCNID;
78   }
79
80   return errorCode;
81 }
```

If you click the icon for a function, the call site is highlighted.

```
147 /* Model step function */
148 void rtwdemo_sil_topmodel_step(void)
149 {
150   /* Logic: '<Root>/Logical Operator2' incorporates:
151    *  Inport: '<Root>/count_enable'
152    *  Inport: '<Root>/counter_mode'
153    *  Logic: '<Root>/Logical Operator'
154    */
155   enableA = ((!rtU.counter_mode) && rtU.count_enable);
156
157   /* Outputs for Enabled SubSystem: '<Root>/CounterTypeA' */
158   CounterTypeA();
159
160   /* End of Outputs for SubSystem: '<Root>/CounterTypeA' */
```

From the Code Execution Profiling Report, you can trace the model component that produces a set of metrics. For example, in the **Section** column, if you click the `CounterTypeA` hyperlink, the Simulink Editor identifies the subsystem.



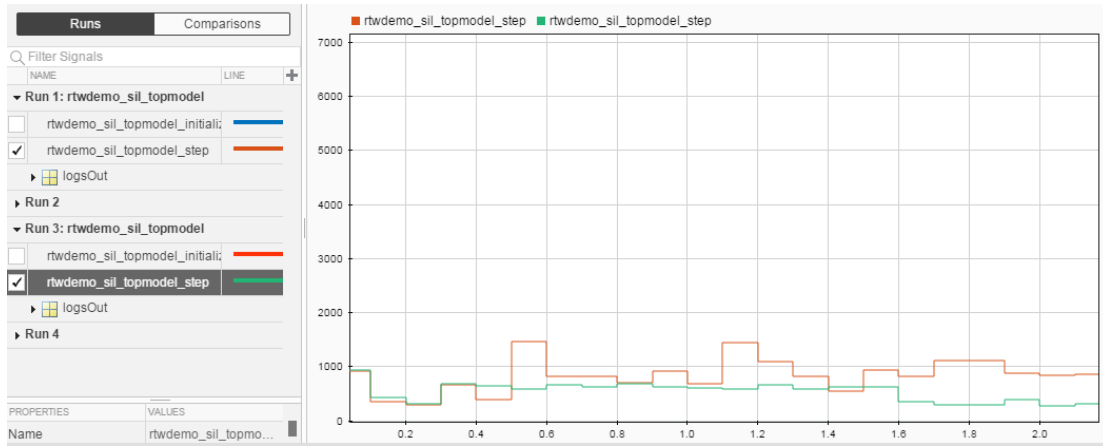
By default, the report displays time in nanoseconds (10^{-9} seconds). You can specify the time unit and numeric display format. For example, to display time in microseconds (10^{-6} seconds), use the following command:

```
>>report(executionProfile,'Units', 'Seconds', 'ScaleFactor', '1e-06', 'NumericFormat', '%0.3f')
```

The report displays time in seconds only if the timer is calibrated, that is, the number of timer ticks per second is known. On a Windows machine, the software determines this value for a SIL simulation. On a Linux machine, calibrate the timer manually. For example, if your processor speed is 1 GHz, specify the number of timer ticks per second:

```
>>executionProfile.TimerTicksPerSecond = 1e9;
```

To display measured execution times for a task or function, click the Simulation Data Inspector icon on the corresponding row. Use the Simulation Data Inspector to manage and compare plots from various simulations.



Note: To observe how code sections are invoked over the execution timeline, use the `timeline` function.

The following table describes the information provided in the code section profiles.

Column	Description
Section	Name of task, top model, subsystem, or Model block. Click the link to go to the model.

Column	Description
	With a task, the sample period and sample offset are listed next to the task name. For example, `rtwdemo_sil_topmodel_step [0.1 0]` indicates that the sample period is 0.1 seconds and the sample offset is 0.
Maximum Execution Time	Longest time between start and end of code section.
Average Execution Time	Average time between start and end of code section.
Maximum Self Time	Maximum execution time, excluding time in child sections.
Average Self Time	Average execution time, excluding time in child sections.
Calls	Number of calls to the code section.
	Icon that you click to see the profiled code section in the Generated Code view of the Code Generation Report. The code section can be a task or a function. The specified workspace variable, for example, `executionProfile`, must be present in the base workspace.
	Icon that you click to display the profiled code section in the Command Window. Equivalent to executing the command `executionProfile.Sections(i)`. The specified workspace variable, for example, `executionProfile`, must be present in the base workspace.
	Icon that you click to display measured execution times with Simulation Data Inspector. The specified workspace variable, for example, `executionProfile`, must be present in the base workspace.

See Also

`annotate` | `report`

Related Examples

- “Code Execution Profiling with SIL and PIL” on page 58-2

- "Analyze Code Execution Data" on page 58-18
- "Simulation Data Inspector in Your Workflow" (Simulink)

More About

- "Tips and Limitations" on page 58-21

Analyze Code Execution Data

After a SIL or PIL simulation, you can analyze execution-time data using methods from the `coder.profile.ExecutionTime` and `coder.profile.ExecutionTimeSection` classes.

1. Open `rtwdemo_sil_topmodel`.
2. On the **Configuration Parameters** > **Code Generation** > **Verification** pane, specify profiling options:
   - Select the **Measure task execution time** check box.
   - Specify a **Workspace variable**, for example, `myExecutionProfile`.
   - From the **Save options** drop-down list, select `All data`.
3. Run a SIL simulation.

The software generates the workspace variable `myExecutionProfile`, an `coder.profile.ExecutionTime` object.

To get the total number of code sections that have profiling data, use the `Sections` method.

```
>> no_of_Sections = myExecutionProfile.Sections

no_of_Sections =

  1×2 ExecutionTimeTaskSection array with properties:

    Name
    Number
    ExecutionTimeInTicks
    SelfTimeInTicks
    TurnaroundTimeInTicks
    TotalExecutionTimeInTicks
    TotalSelfTimeInTicks
    TotalTurnaroundTimeInTicks
    MaximumExecutionTimeInTicks
    MaximumExecutionTimeCallNum
    MaximumSelfTimeInTicks
    MaximumSelfTimeCallNum
    MaximumTurnaroundTimeInTicks
    MaximumTurnaroundTimeCallNum
    NumCalls
    ExecutionTimeInSeconds
    Time
```

To get the `coder.profile.ExecutionTimeSection` object for a profiled code section, use the method `Sections`.

```
>> FirstSectionProfile = myExecutionProfile.Sections(1)
SecondSectionProfile = myExecutionProfile.Sections(2)

FirstSectionProfile =

  ExecutionTimeTaskSection with properties:

                           Name: 'rtwdemo_sil_topmodel_initialize'
                         Number: 1
           ExecutionTimeInTicks: 1188
                SelfTimeInTicks: 1188
            TurnaroundTimeInTicks: 1188
       TotalExecutionTimeInTicks: 1188
            TotalSelfTimeInTicks: 1188
      TotalTurnaroundTimeInTicks: 1188
     MaximumExecutionTimeInTicks: 1188
     MaximumExecutionTimeCallNum: 1
          MaximumSelfTimeInTicks: 1188
          MaximumSelfTimeCallNum: 1
    MaximumTurnaroundTimeInTicks: 1188
    MaximumTurnaroundTimeCallNum: 1
                      NumCalls: 1
          ExecutionTimeInSeconds: 5.4000e-07
                           Time: 0


SecondSectionProfile =

  ExecutionTimeTaskSection with properties:

                           Name: 'rtwdemo_sil_topmodel_step [0.1 0]'
                         Number: 2
           ExecutionTimeInTicks: [1×101 uint64]
                SelfTimeInTicks: [1×101 uint64]
            TurnaroundTimeInTicks: [1×101 uint64]
       TotalExecutionTimeInTicks: 70316
            TotalSelfTimeInTicks: 70316
      TotalTurnaroundTimeInTicks: 70316
     MaximumExecutionTimeInTicks: 2448
     MaximumExecutionTimeCallNum: 2
          MaximumSelfTimeInTicks: 2448
          MaximumSelfTimeCallNum: 2
    MaximumTurnaroundTimeInTicks: 2448
    MaximumTurnaroundTimeCallNum: 2
                      NumCalls: 101
          ExecutionTimeInSeconds: [1×101 double]
                           Time: [101×1 double]
```

Use `coder.profile.ExecutionTimeSection` methods to extract profiling information for a particular code section. For example, use `Name` to obtain the name of a profiled task.

```
>> name_of_section = SecondSectionProfile.Name

name_of_section =
```

```
rtwdemo_sil_topmodel_step [0.1 0]
```

If the timer is uncalibrated and you know the timer rate, for example 2.2 GHz, you can use the `coder.profile.ExecutionTime` method `TimerTicksPerSecond` to calibrate the timer:

```
>> myExecutionProfile.TimerTicksPerSecond = 2.2e9;
>> SecondSectionProfile = myExecutionProfile.Sections(2);
```

Related Examples

- “Code Execution Profiling with SIL and PIL” on page 58-2
- “View and Compare Code Execution Times” on page 58-7
- “Tips and Limitations” on page 58-21

Tips and Limitations

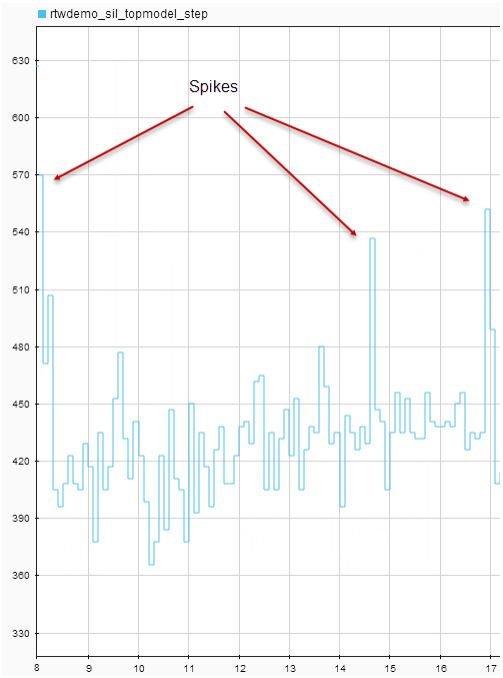
In this section...

Triggered Model Block

Consider the case where a triggered Model block is configured to run in the SIL or PIL simulation mode. The software generates one execution-time measurement each time the referenced model is triggered to run. If there are multiple triggers in a single time step, the software generates multiple measurements for the triggered Model block. Conversely, if there is no trigger in a given time step, the software generates no time measurements.

Outliers in Execution-Time Profiles

When you run a SIL simulation with execution time profiling enabled, you might see spikes in execution-time measurements.



The spikes are due to process preemption that occurs with a multitasking host operating system. If the operating system preempts the SIL process and runs another process, the measured execution time includes the time during which the SIL process is suspended. With a PIL simulation, you do not see spikes because code execution on the target is not preempted.

Counter wrapping produces execution-time measurements that are smaller than expected. For SIL, the counter wraps when an execution-time period is greater than 2^{64} ticks (2^{32} ticks if the MEX compiler is LCC). For PIL, the wrapping point depends on the timer you specify and can be 2^{8}, 2^{16}, 2^{32}, or 2^{64} ticks.

Consider a PIL example where the timer frequency is 20 MHz. For a 32-bit timer, wrapping occurs when the execution-time period is greater than `1/(20e6)*(2^32-1)`, that is, 214.7 s. However, for a 16-bit timer, the point at which wrapping occurs is 0.0033 s.

For a real-time, multi-core application, the software accommodates synchronization discrepancies when recording timer values for different cores, which effectively reduces the timer measurement range.

Hardware-Specific Timer

If your target configuration does not already specify a timer, create a timer object that provides details of the hardware-specific timer and associated source files:

- For SIL simulation, the timer word length is 64 bits.
- For PIL simulation, you can specify an 8-, 16-, 32-, or 64-bit timer.

Task Context Switching Due to Preemption

Profiling instrumentation is intrusive and affects the quantity that it is meant to measure. Therefore, the design goal is to maximize code understanding with a minimum of instrumentation. For example, with a real-time system, there can be task context switches due to preemption. These context switches are not explicitly instrumented. To record the start and end of each task, the software must infer context switches from instrumentation. As a result, the software reports behavior that is an estimate. The estimate is subject to error because of incomplete instrumentation within the kernel.

In some cases, when the software cannot accurately determine behavior, the software generates a warning:

`Warning: Analysis unsuccessful for one or more profiling data points. ...`
For example, the software might generate this warning when not all mutex take system calls (associated with rate transitions) are instrumented. In the case of Simulink Real-Time, this situation might arise if you generate code for a model reference hierarchy without enabling function profiling for all referenced models (`set_param(model, 'CodeProfilingInstrumentation', 'on')`). If a mutex take system call is not instrumented, a task context switch might occur that is not visible to the execution profiling analysis.

In other cases, although the software cannot accurately determine behavior, the software does *not* generate a warning.

Data Type Replacement Support

Data type replacement does not support the measurement of function execution times. For your model, clear one of the following check boxes:

- **Configuration Parameters > Code Generation > Verification > Measure function execution times**
- **Configuration Parameters > Code Generation > Data Type Replacement > Replace data type names in the generated code**

Subsystem Code Reuse

You cannot generate execution-time profiles for function call sites within subsystem code that is reused across a model or multiple models. For information about subsystem code reuse, see "Code Reuse For Subsystems Shared Across Models" (Simulink Coder).

Cannot Load Execution-Time Measurements from Previous Release

You cannot load execution-time measurements saved with a previous release. For example, using R2014a, you save workspace variables to a MAT-file. One of the workspace variables contains execution-time measurements. In R2015b, if you try to load the MAT-file, you see this error:

```
Format of execution profiling data is invalid. This error can occur if
you load data from a previous release. Loading data from a previous
release is not supported.
```

Code Execution Profiling for MATLAB Coder

Execution Time Profiling for SIL and PIL

During a software-in-the-loop (SIL) or processor-in-the-loop (PIL) execution, you can produce a profile of execution times for code generated from entry-point functions. The software calculates execution times from data that is obtained through instrumentation probes added to the SIL or PIL application.

Use the execution time profile to check whether your code runs within the required time on your target hardware:

- If code execution overruns, look for ways to reduce execution time.
- If your code easily meets time requirements, consider enhancing functionality to exploit the unused processing power.

At the end of the SIL or PIL execution, you can:

- View a report of code execution times.
- Use the Simulation Data Inspector to view and compare plots of function execution times.
- Access and analyze execution time profiling data.

Note: SIL and PIL execution supports multiple entry-point functions. An entry-point function can call another entry-point function as a subfunction. However, the software generates execution time profiles only for functions that are called at the entry-point level. The software does not generate execution time profiles for entry-point functions that are called as subfunctions by other entry-point functions.

Related Examples

- "Generate Execution Time Profile" on page 59-3
- "View Execution Times" on page 59-4
- "Analyze Execution Time Data" on page 59-7

Generate Execution Time Profile

Before running a software-in-the-loop (SIL) or processor-in-the-loop (PIL) execution, enable execution time profiling:

1. To open the MATLAB Coder app, on the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the app icon.
2. To open your project, click  and then click `Open existing project`. Select the project.
3. On the **Generate Code** page, click **Verify Code**.
4. Select the **Enable entry point execution profiling for SIL/PIL** check box.

Or, from the Command Window, specify the `CodeExecutionProfiling` property of your `coder.EmbeddedCodeConfig` object. For example:

```
config.CodeExecutionProfiling = true;
```

Related Examples

- "Software-in-the-Loop Execution with the MATLAB Coder App" on page 66-4
- "Processor-in-the-Loop Execution with the MATLAB Coder App" on page 66-25
- "View Execution Times" on page 59-4
- "Analyze Execution Time Data" on page 59-7

More About

- "Execution Time Profiling for SIL and PIL" on page 59-2

View Execution Times

When you run a SIL or PIL execution with execution time profiling enabled, the software generates a message in the **Test Output** tab. For example:

```
Current plot held
### Starting SIL execution for 'kalman01'
    To terminate execution: clear kalman01_sil
    Execution profiling data is available for viewing. Open Simulation Data Inspector.
    Execution profiling report available after termination.
Current plot released
```

To observe streamed execution times while the execution runs, click the `Simulation Data Inspector` link.

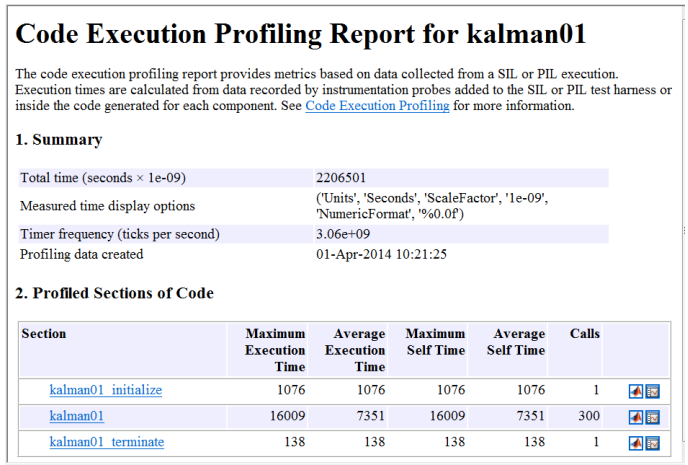
To open the code execution profiling report:

1. Click the `Stop SIL Verification` link.

   The software terminates the execution process and displays a new link.

   ```
   ### Stopping SIL execution for 'kalman01'
       Execution profiling report: report(getCoderExecutionProfile('kalman01'))
   ```

2. Click the new link.



Code Execution Profiling Report for kalman01

The code execution profiling report provides metrics based on data collected from a SIL or PIL execution. Execution times are calculated from data recorded by instrumentation probes added to the SIL or PIL test harness or inside the code generated for each component. See Code Execution Profiling for more information.

1. Summary

Total time (seconds × 1e-09)	2206501
Measured time display options	('Units', 'Seconds', 'ScaleFactor', '1e-09', 'NumericFormat', '%0.0f')
Timer frequency (ticks per second)	3.06e+09
Profiling data created	01-Apr-2014 10:21:25

2. Profiled Sections of Code

Section	Maximum Execution Time	Average Execution Time	Maximum Self Time	Average Self Time	Calls	
kalman01_initialize	1076	1076	1076	1076	1	
kalman01	16009	7351	16009	7351	300	
kalman01_terminate	138	138	138	138	1	

The first section provides a summary. The second section contains information about profiled code sections.

The report contains time measurements for:

- The *entry_point_fn*`_initialize` function, for example, `kalman01_initialize`.
- The entry-point function, for example, `kalman01`.
- The *entry_point_fn*`_terminate` function, for example, `kalman01_terminate`.

By default, the report displays time in nanoseconds (10^{-9} seconds). You can specify the time unit and numeric display format. For example, to display time in microseconds (10^{-6} seconds), use the `report` command:

```
executionProfile=getCoderExecutionProfile('kalman01'); % Create workspace var
report(executionProfile, ...
       'Units', 'Seconds', ...
       'ScaleFactor', '1e-06', ...
       'NumericFormat', '%0.3f')
```

The report displays time in seconds only if the timer is calibrated, that is, the number of timer ticks per second is established. On a Windows machine, the software determines this value for a SIL simulation. On a Linux machine, you must manually calibrate the timer. For example, if your processor speed is 1 GHz, specify the number of timer ticks per second:

```
executionProfile.TimerTicksPerSecond = 1e9;
```

To display measured execution times for a code section, click the Simulation Data Inspector icon  on the corresponding row. You can use the Simulation Data Inspector to manage and compare plots from various executions.

The following table lists the information provided in the code section profiles.

Column	Description
Section	Name of function from which code is generated.
Maximum Execution Time	Longest time between start and end of code section.
Average Execution Time	Average time between start and end of code section.
Maximum Self Time	Maximum execution time, excluding time in child sections.
Average Self Time	Average execution time, excluding time in child sections.

Column	Description
Calls	Number of calls to the code section.
	Icon that you click to display the profiled code section.
	Icon that you click to display measured execution times with Simulation Data Inspector.

Related Examples

- "Generate Execution Time Profile" on page 59-3
- "Analyze Execution Time Data" on page 59-7
- "Simulation Data Inspector in Your Workflow" (Simulink)

Analyze Execution Time Data

After a software-in-the-loop (SIL) or processor-in-the-loop (PIL) execution, you can analyze execution-time data using methods from the `coder.profile.ExecutionTime` and `coder.profile.ExecutionTimeSection` classes.

In the following example, you run a SIL execution and apply supplied methods to execution-time data.

Extract Execution Time Data for Kalman Estimator Code

1 Run SIL execution to generate execution time data

Copy MATLAB code to your working folder.

```
src_dir = ...
    fullfile(docroot,'toolbox','coder','examples','kalman');

copyfile(fullfile(src_dir,'kalman01.m'), '.')
copyfile(fullfile(src_dir,'test01_ui.m'), '.')
copyfile(fullfile(src_dir,'plot_trajectory.m'), '.')
copyfile(fullfile(src_dir,'position.mat'), '.')
```

For a description of the Kalman estimator, see "C Code Generation at the Command Line" (MATLAB Coder).

Create a coder.EmbeddedCodeConfig object.

```
config = coder.config('lib');
config.GenerateReport = true; % HTML report
```

Configure the object for SIL and enable execution time profiling.

```
config.VerificationMode = 'SIL';
config.CodeExecutionProfiling = true;
```

Generate library code for the `kalman01` MATLAB function and the SIL interface.

```
codegen('-config', config, '-args', {zeros(2,1)}, 'kalman01');
```

Run the MATLAB test file `test01_ui` with `kalman01_sil`. `kalman01_sil` is the SIL interface for `kalman01`.

```
coder.runTest('test01_ui', ['kalman01_sil.' mexext]);
```

You see the following message.

```
### Starting SIL execution for 'kalman01'
    To terminate execution: clear kalman01_sil
    Execution profiling data is available for viewing. Go to Simulation Data Inspector.
    Execution profiling report available after termination.
Current plot released
```

Terminate the SIL execution process. Click the link `clear kalman01_sil`.

```
 ### Stopping SIL execution for 'kalman01'
    Execution profiling report: report(getCoderExecutionProfile('kalman01'))
```

2 Create workspace variable that holds execution time data

Use the `getCoderExecutionProfile` function to create a workspace variable that holds execution time profiling data.

```
executionProfile=getCoderExecutionProfile('kalman01');
```

3 Extract code sections

Use the `Sections` method.

```
allSections = executionProfile.Sections
```

The software displays the number of code sections and a list of properties.

```
allSections =

  1x3 ExecutionTimeTaskSection array with properties:

    Name
    Number
    ExecutionTimeInTicks
    SelfTimeInTicks
    TurnaroundTimeInTicks
    TotalExecutionTimeInTicks
    TotalSelfTimeInTicks
    TotalTurnaroundTimeInTicks
    MaximumExecutionTimeInTicks
    MaximumExecutionTimeCallNum
    MaximumSelfTimeInTicks
    MaximumSelfTimeCallNum
    MaximumTurnaroundTimeInTicks
    MaximumTurnaroundTimeCallNum
    NumCalls
    ExecutionTimeInSeconds
    Time
```

4 Extract execution time data from specific code section

Specify the code section that you want to examine.

```
secondSectionProfile = executionProfile.Sections(2)
```

The software displays profile data for the code section.

```
secondSectionProfile =

  ExecutionTimeTaskSection with properties:

                          Name: 'kalman01'
                        Number: 2
          ExecutionTimeInTicks: [1x300 uint64]
               SelfTimeInTicks: [1x300 uint64]
         TurnaroundTimeInTicks: [1x300 uint64]
      TotalExecutionTimeInTicks: 6641016
           TotalSelfTimeInTicks: 6641016
     TotalTurnaroundTimeInTicks: 6641016
    MaximumExecutionTimeInTicks: 48864
    MaximumExecutionTimeCallNum: 158
         MaximumSelfTimeInTicks: 48864
         MaximumSelfTimeCallNum: 158
   MaximumTurnaroundTimeInTicks: 48864
   MaximumTurnaroundTimeCallNum: 158
                      NumCalls: 300
         ExecutionTimeInSeconds: [1x300 double]
                          Time: [300x1 double]
```

You can extract specific properties, for example, the name of a profiled function.

```
nameOfSection = secondSectionProfile.Name
```

The software displays the name.

```
nameOfSection =

kalman01
```

The following table lists the information that you can extract from each code section.

Property	Description
`Name`	Name of entry-point function
`Number`	Code section number

Property	Description
`ExecutionTimeInTicks`	Vector of execution times, measured in timer ticks. Each element contains the difference between the timer reading at the start and at the end of the code section. The data type is the same data type as the data type of the timer used on the target, which allows you to infer the maximum range of the timer measurements.
`SelfTimeInTicks`	Vector of timer tick numbers. Each element contains the number of ticks recorded for the code section, excluding the time spent in calls to child functions.
`TurnaroundTimeInTicks`	Vector of timer tick numbers. Each element contains the number of ticks recorded between the start and the finish of the code section. Unless the code is preempted, this number is the same number as the execution time.
`TotalExecutionTimeInTicks`	Total number of timer ticks recorded for the code section over the entire execution.
`TotalSelfTimeInTicks`	Total number of timer ticks recorded for the profiled code section over the entire execution. However, this number excludes the time spent in calls to child functions.
`TotalTurnaroundTimeInTicks`	Total number of timer ticks recorded between the start and the finish of the profiled code section over the entire execution. Unless the code is preempted, this number is the same as the total execution time.
`MaximumExecutionTimeInTicks`	Maximum number of timer ticks recorded for a single invocation of the code section over the execution.
`MaximumExecutionTimeCallNum`	Number of call in which `MaximumExecutionTimeInTicks` occurs.
`MaximumSelfTimeInTicks`	Maximum number of timer ticks recorded for a single code section invocation, but excluding the time spent in calls to child functions.
`MaximumSelfTimeCallNum`	Number of call in which `MaximumSelfTimeInTicks` occurs.
`MaximumTurnaroundTimeInTicks`	Maximum number of timer ticks recorded between the start and the finish of a single invocation of the profiled code section over the execution. Unless the code is preempted, this time is the same as the maximum execution time.

Property	Description
`MaximumTurnaroundTimeCallNum`	Number of call in which `MaximumTurnaroundTimeInTicks` occurs.
`NumCalls`	Total number of calls to the code section over the entire execution.
`ExecutionTimeInSeconds`	Vector of execution times, measured in seconds. Each element contains the difference between the timer reading at the start and at the end of the code section. Produced only if `TimerTicksPerSecond` is set.
`Time`	Vector of execution time measurements for the code section.

Related Examples

- "Generate Execution Time Profile" on page 59-3
- "View Execution Times" on page 59-4
- "Simulation Data Inspector in Your Workflow" (Simulink)

Verification

Simulation and Code Comparison in Simulink Coder

Simulation and Code Comparison

In this section...

This example shows how to verify the answers computed by code generated from the `slexAircraftExample` model. It shows how to capture and compare two sets of output data. Simulating the model produces one set of output data. Executing the generated code produces a second set of output data.

Note To obtain a valid comparison between model output and the generated code, use the same **Solver options** and **Step size** for the simulation run and the build process.

Configure Signal Data for Logging

Configure the model for logging and recording signal data.

1. Make sure that `slexAircraftExample` is closed. Clear the base workspace to eliminate the results of previous simulation runs. In the Command Window, type:

   ```
   clear
   ```

   The clear operation clears variables created during previous simulations and all workspace variables, some of which are standard variables that the `slexAircraftExample` model requires.
2. In the Command Window, enter `slexAircraftExample` to open the model.
3. In the model window, choose **File > Save As**, navigate to the working folder, and save a copy of the `slexAircraftExample` model as `myAircraftExample`.
4. Set up your model to log signal data for signals: `Stick`, `alpha,rad`, and `q, rad/sec`. For each signal:

   a. Right-click the signal. From the context menu, select **Properties**.

b In the Signal Properties dialog box, select **Log signal data**.

c In the **Logging name** section, from the drop-down list, select `Custom`.

d In the text field, enter the logging name for the corresponding signal.

Signal Name	Logging Name
`Stick`	`Stick_input`
`alpha,rad`	`Alpha`
`q, rad/sec`	`Pitch_rate`

e Click **Apply** and **OK**.

For more information, see "Export Signal Data Using Signal Logging" (Simulink).

5 Select **Simulation** > **Model Configuration Parameters** to open the Configuration Parameters dialog box.

6 Select the **Solver** pane and in the **Solver options** section, specify the **Type** parameter as `Fixed-step`.

7 On the **Data Import/Export** pane:

- Specify the **Format** parameter as `Structure with time`.
- Clear the **States** parameter check box.
- Select the **Signal logging** parameter.
- Select the **Record logged workspace data in Simulation Data Inspector** parameter to enable logged signal data to send to the Simulation Data Inspector after the simulation is finished.

8 Save the model.

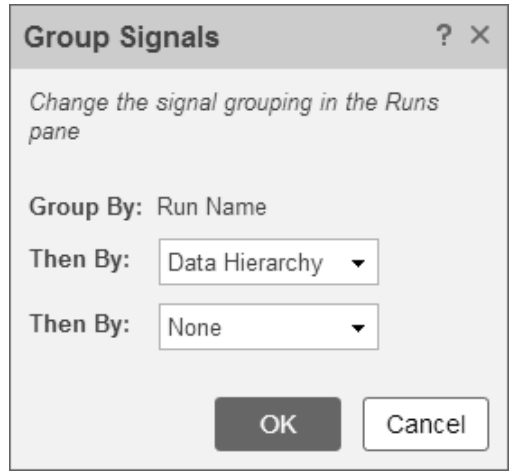
Proceed to "Log Simulation Data" on page 60-3.

Log Simulation Data

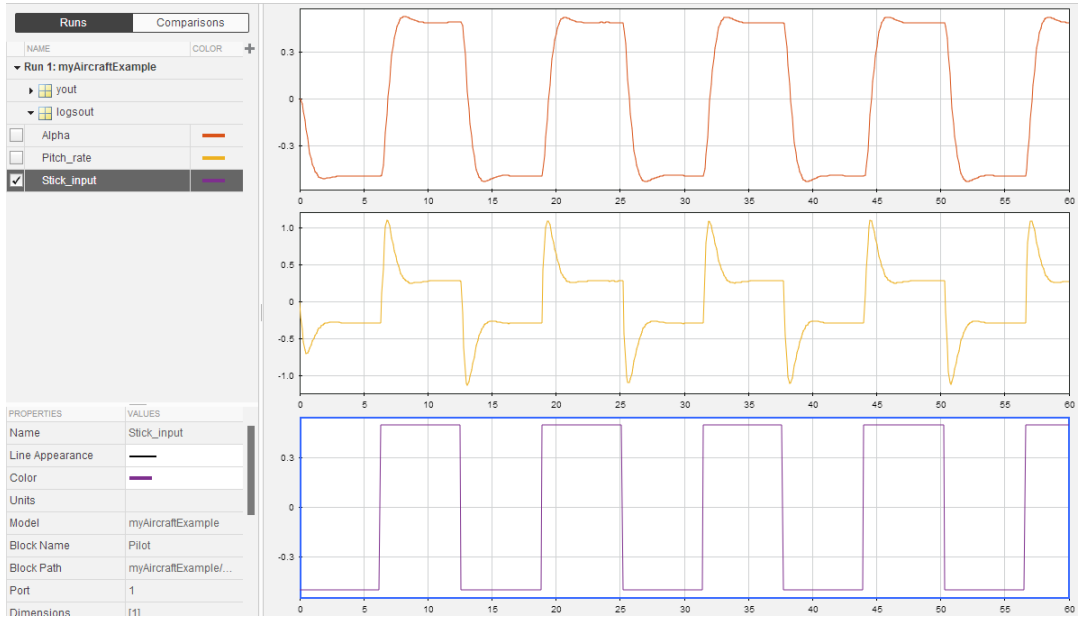
Run the simulation, log the signal data, and view the data in the Simulation Data Inspector.

1 Run the model. When the simulation is done, on the Simulink Editor toolbar, the **Simulation Data Inspector** button is highlighted to indicate that new simulation output is available in the Simulation Data Inspector.

2. Click the **Simulation Data Inspector** button to open the Simulation Data Inspector.
3. Group the signals:
   a. On the **Visualize** tab, click **Group Signals**.
   b. In the Group Signals dialog box, select `Data Hierarchy` from the **Then By** list.



   c. Click **OK**.
4. Click the `logsout` expander to view the logged signals.
5. Click the **Format** tab.
6. Click the **Subplots** button and select `3x1` to show three subplots.
7. For each signal:
   a. Click the top subplot. A blue border indicates the plot selection.
   b. Select the check box next to the `Alpha` signal name. The signal data appears in the subplot.
   c. Plot the `Pitch_rate` signal in the middle subplot.
   d. Plot the `Stick_input` signal in the bottom subplot.



Proceed to “Run Executable and Load Data” on page 60-5.

Run Executable and Load Data

You must rebuild and run the `myAircraftExample` executable to obtain a valid data file because you have modified the model.

1. Select **Simulation > Model Configuration Parameters** to open the Configuration Parameters dialog box.
2. Select the **Code Generation > Interface** pane.
3. Set the **MAT-file variable name modifier** parameter to `rt_`. `rt_` is prefixed to each variable that you selected for logging in the first part of this example.
4. Click **Apply** and **OK**.
5. Save the model.
6. On the Simulink Editor toolbar, click the **Build Model** button to generate code.
7. When the build is finished, run the standalone program from the Command Window.

```
!myAircraftExample
```

The executing program writes the following messages to the Command Window.

```
** starting the model **
** created myAircraftExample.mat **
```

8 Load the data file `myAircraftExample.mat`.

```
load myAircraftExample
```

Tip: For UNIX platforms, run the executable in the Command Window with the syntax `!./executable_name`. If preferred, run the executable from an OS shell with the syntax `./executable_name`. For more information, see "Run External Commands, Scripts, and Programs" (MATLAB).

Proceed to "Visualize and Compare Results" on page 60-6.

Visualize and Compare Results

When you follow the example sequence that began in "Configure Signal Data for Logging" on page 60-2, you obtain data from a Simulink run of the model and from a run of the program generated from the model.

1 To view the execution output for `alpha,rad`, import the data into the Simulation Data Inspector.

   a On the Simulation Data Inspector **Visualize** tab, click the **Import** button to open the Import dialog.

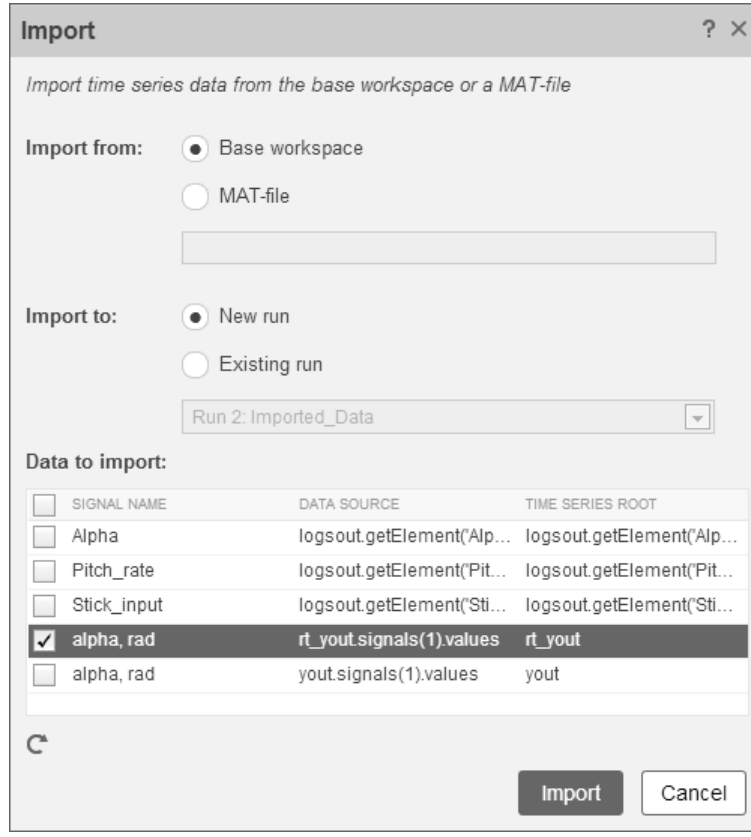
   b Specify **Import from** as **Base workspace**.

   c Specify **Import to** as **New run**.

   d To the left of **Signal Name**, click the check mark to clear the check boxes.

   e Select the check box for the `alpha,rad` data where the **Time Series Root** is `rt_yout`.

   f Click **Import**.



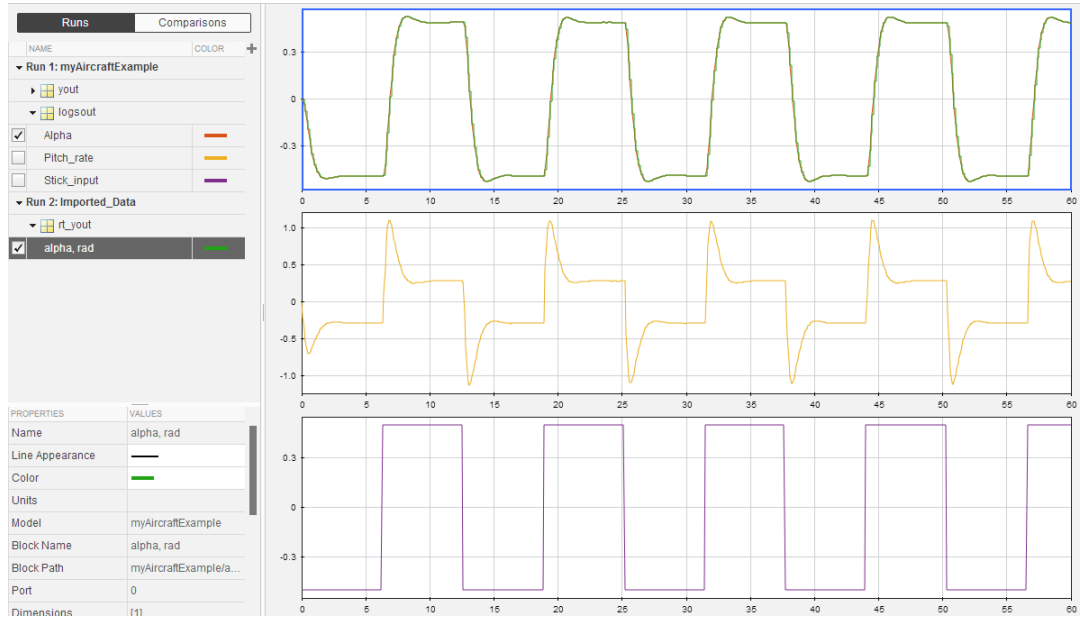
The selected data is now under **Run 2: Imported_Data**.

2 View a plot of the executed data.

a Click the `rt_yout` expander.

b Click the top subplot and select the check box next to the `alpha, rad` signal name. The signal data appears in the top subplot.

The `alpha, rad` signal from Run 1 and Run 2 overlap in the subplot because the signals are equivalent.



It is possible to see a very small difference between simulation and code generation results. A slight difference can be caused by many factors, including:

- Different compiler optimizations
- Statement ordering
- Run-time libraries

For example, a function call such as `sin(2.0)` can return a slightly different value depending on which C library you use. Such variations can also cause differences between your results and these results.

Compare States for Simulation and Code Generation

The order in which Simulink logs states during simulation is different than the order in which Simulink Coder logs states during code generation. If you want to compare states between simulation and code generation, sort the states by block name.

For example, by default, Simulink exports state data to the MATLAB variable, `xout`. Simulink Coder exports state data to the variable `rt_xout`. To sort the state data for these variables, enter the following commands in the MATLAB Command Window:

```
[~,idx1]=sort({xout.signals.blockName});
xout_sorted=[xout.signals(idx1).values];
[~,idx2]=sort({rt_xout.signals.blockName});
rt_xout_sorted=[rt_xout.signals(idx2).values];
```

You can confirm that the logging order is the same between code generation and simulation by entering the following command in the MATLAB Command Window:

```
isequal(xout_sorted, rt_xout_sorted)
```

Related Examples

- "Log Program Execution Results" (Simulink Coder)

Code Tracing in Embedded Coder

What Is Code Tracing?

Code tracing (traceability) involves using hyperlinks to navigate between a line of generated code and its corresponding objects in a model. You can also right-click an object or objects in a model to find the lines of code to which they correspond. This two-way navigation is *bidirectional* traceability.

Code tracing provides a way to:

- Verify generated code. You can identify which model objects correspond to a line of code. You are able to keep track of code from different objects that you have or have not reviewed.
- Include comments in code generated for large-scale models. You can identify objects in generated code and avoid manually entering comments or descriptions.

The HTML code generation report that the code generator produces for a model includes resources that support code tracing:

- Code element hyperlinks (indicated with underlining) to trace through and toggle between generated source and header files.
- Tags in code comments that identify objects in a model from which lines of code are generated.
- Line number hyperlinks which link to the model component from which the line of code was generated.

Traceable Objects

Bidirectional traceability is supported for blocks and the following Stateflow objects:

- States
- Transitions
- MATLAB functions (not supported for external code called from a MATLAB function)

  Note: Traceability is not supported for external code that you call from a MATLAB function.
- Truth Table blocks and truth table functions
- Graphical functions

- Simulink functions
- State transition tables

Traceability in one direction is supported for these Stateflow objects:

- Events (code-to-model)

  Code-to-model traceability works for explicit events, but not implicit events. Clicking a hyperlink for an explicit event in the generated code highlights that item in the **Contents** pane of the Model Explorer.
- Junctions (model-to-code)

  Model-to-code traceability works for junctions with at least one outgoing transition. Right-clicking such a junction in the Stateflow Editor highlights the line of code that corresponds to the first outgoing transition for that junction.

Note: MATLAB Function blocks that you insert directly in a Simulink model are also traceable. For more information, see "Use Traceability in MATLAB Function Blocks" (Simulink).

Workflow Traceability

The basic workflow for using traceability is:

1. Open your model.
2. Define your system target file as an embedded real-time (`ert`) target.
3. Enable and configure the traceability options.
4. Generate the source code and header files for your model.
5. Do one or both of these steps:
   - Trace a line of generated code to the model.
   - Trace objects in the model to lines of code.

Related Examples

- "Trace Code to Model Objects by Using Hyperlinks" on page 61-6
- "Trace Model Objects to Generated Code" on page 61-8

- "Reload Existing Traceability Information" on page 61-28
- "Customize Traceability Reports" on page 61-29

More About

- "Traceability Tags" on page 61-5

Traceability Tags

A traceability tag appears in a comment above the corresponding line of generated code. The format of the tags is `<system>/block_name`.

- *system* is one of the following:
  - The text `Root`
  - A unique system number assigned by the Simulink engine
- *block_name* is the name of the source block

The code generator documents the tags for a model in the comments section of the generated header file *model*`.h`. For example, the following comment appears in the header file for a model, `foo`, that has a subsystem `Outer` and a nested subsystem `Inner`:

```
/* Here is the system hierarchy for this model.
 *
 * <Root> : foo
 * <S1>   : foo/Outer
 * <S2>   : foo/Outer/Inner
 */
```

This code shows a tag comment adjacent to a line of code. A Gain block at the root level of a source model generates this code:

```
/* Gain: '<Root>/UnDeadGain1' */
rtb_UnDeadGain1_h = dead_gain_U.In1 *
  dead_gain_P.UnDeadGain1_Gain;
```

The following code shows a tag comment adjacent to a line of code. A Gain block within a subsystem one level below the root level of the source model generates this code:

```
/* Gain: '<S1>/Gain' */
dead_gain_B.tempO *= (dead_gain_P.s1_Gain_Gain);
```

Trace Code to Model Objects by Using Hyperlinks

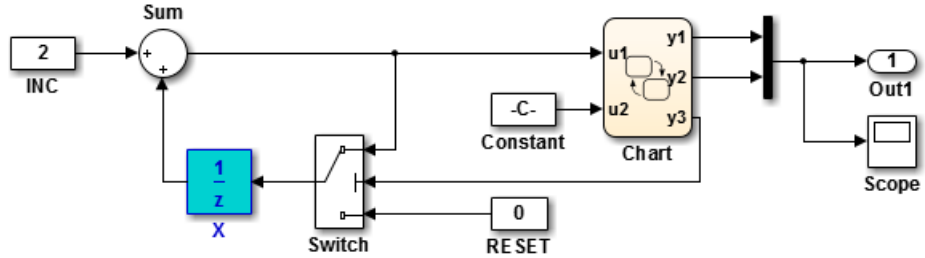
When using the Simulink Coder product, you can trace code to model objects by using the `hilite_system` command. The Embedded Coder product simplifies traceability with the use of hyperlinks in HTML code generation reports. The reports display hyperlinks in comment lines in generated code. To highlight the corresponding block or subsystem in the model diagram, click the hyperlinks.

To use hyperlinks for tracing code to model objects:

1. Open the model and make sure it is configured for an ERT target.
2. In the Configuration Parameters dialog box, on the **Code Generation** > **Report** pane, the **Create code generation report** parameter is selected by default. When selected, this parameter enables and selects **Open report automatically** and **Code-to-model**.
3. Build or generate code for the model. An HTML code generation report is displayed.
4. In the HTML report window, click hyperlinks to highlight source blocks. For example, generate an HTML report for model `rtwdemo_hyperlinks`. In the generated code for the model step function in `rtwdemo_hyperlinks.c`, click the first UnitDelay block hyperlink.

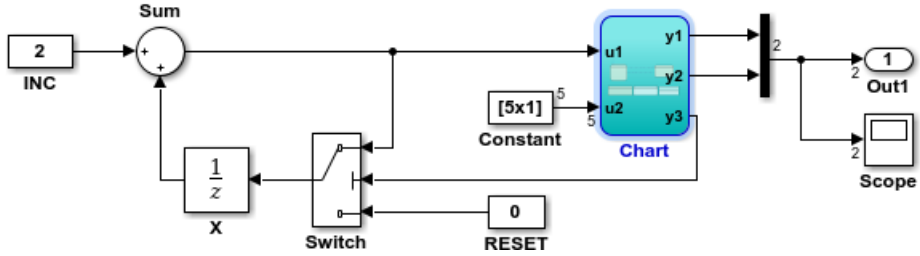
```
134  /* Model step function */
135  void rtwdemo_hyperlinks_step(void)
136  {
137    /* Chart: '<Root>/Chart' incorporates:
138     *  Constant: '<Root>/Constant'
139     *  Constant: '<Root>/INC'
140     *  Sum: '<Root>/Sum'
141     */
142    /* Gateway: Chart */
143    if (rtDWork.temporalCounter_i1 < 7U) {
144      rtDWork.temporalCounter_i1++;
145    }
```

In the model window, the corresponding UnitDelay block is highlighted.



To use line numbers for tracing code to model objects:

1. In the previous example, `rtwdemo_hyperlinks`, click the hyperlink at line number 144.
2. In the model window, the Chart subsystem is highlighted and contains the operation on line 144.



Trace Model Objects to Generated Code

1. Open the model and make sure that it is configured for an ERT target.
2. In the Configuration Parameters dialog box, the **Code Generation > Report > Create code generation report** parameter is selected by default. When selected, the parameter enables and selects the **Open report automatically** and **Code-to-model** parameters.
3. On the **All Parameters** tab, select **Model-to-code**. This parameter:
   - Enables the **Configure** button, which opens a dialog box for loading existing trace information.
   - Enables and selects parameters for customizing the content of a traceability report.
4. Build or generate code for the model. An HTML code generation report is displayed.
5. In the model window, right-click a model object. To select multiple blocks, hold **Shift** and right-click additional blocks.
6. From the context menu, select **C/C++ Code > Navigate to C/C++ Code**. In the HTML code generation report, you see the first instance of highlighted code that is generated for the model object. In the left pane of the report, numbers that appear to the right of generated file names indicate the total number of highlighted lines in each file. The following figure shows the result of tracing the Unit Delay block in model `rtwdemo_hyperlinks`.

At the top of the code window, use the navigation bar to move forward and backward through multiple instances of highlighted lines. Use the navigation sidebar to go directly to a line of code.

If you close and reopen a model, the **Navigate to Code** context menu option might not be available because Embedded Coder cannot find a build folder for your model in the current working folder. Do one of the following:

- Reset the current working folder to the parent folder of the existing build folder.
- Select **Model-to-code** and rebuild the model. Rebuilding the model regenerates the build folder into the current working folder.
- Click **Configure**. In the Model-to-code navigation dialog box, reload the existing trace information.

See Also

"Model-to-code" (Simulink Coder) | "Configure" (Simulink Coder)

Trace Stateflow Objects in Generated Code

In this section...

Bidirectional Traceability for States and Transitions

See how bidirectional traceability works for states and transitions by following these steps:

1. At the command prompt, type `old_sf_car`.
2. Open the Model Configuration Parameters dialog box.
3. In the **Code Generation** pane, go to the **Target selection** section and enter `ert.tlc` for the system target file. Click **Apply** in the lower-right corner of the window. Traceability comments appear in generated code only for embedded real-time targets.
4. In the **Code Generation** > **Report** pane, the **Create code generation report** parameter is selected by default. This step automatically selects **Open report automatically** and **Code-to-model**.
5. On the **All Parameters** tab, select **Model-to-code**. Click **Apply**.
6. Go to the **Code Generation** > **Interface** pane. In the **Software environment** section, select **continuous time**. Click **Apply**. Before generating code, you must perform this step because this model contains a block with a continuous sample time.
7. Press **Ctrl+B**.

   This step generates source code and header files for the `old_sf_car` model that contains the `shift_logic` chart. After the code generation process is complete, the code generation report appears.
8. In the report, click the `old_sf_car.c` hyperlink.

9 To see the traceability comments, scroll down through the code. The following line numbers can differ from the numbers that appear in your code generation report.

```
185 /* Function for Chart: '<Root>/shift_logic' */
186 static void old_sf_car_gear_state(const int32_T *sfEvent)
187 {
188   /* During 'gear_state': '<S5>:2' */
189   switch (old_sf_car_DW.is_gear_state) {
190    case old_sf_car_IN_first:
191     /* During 'first': '<S5>:6' */
192     if (*sfEvent == old_sf_car_event_UP) {
193       /* Transition: '<S5>:12' */
194       old_sf_car_DW.is_gear_state = old_sf_car_IN_second;
195
196       /* Entry 'second': '<S5>:4' */
197       old_sf_car_B.gear = 2.0;
```

10 To navigate to the corresponding model component, click the line number hyperlinks.

11 Click the `<S5>:2` hyperlink in this traceability comment:

```
/* During 'gear_state': '<S5>:2' */
```

The corresponding state appears highlighted in the chart.

12 Click the `<S5>:12` hyperlink in this traceability comment:

```
/* Transition: '<S5>:12' */
```

The corresponding transition appears highlighted in the chart. To remove highlighting from an object in the chart, select **Display** > **Remove Highlighting**.

13 You can also trace objects in the model to lines of generated code. In the chart, right-click the object `gear_state` and select **C/C++ Code** > **Navigate to C/C++ Code**.

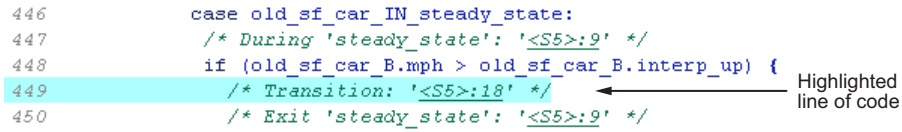
The code for that state appears highlighted in `old_sf_car.c`.

```
188   /* Function for Stateflow: '<Root>/shift_logic' */
189   static void old_sf_car_gear_state(void)
190   {
191     /* During 'gear_state': '<S5>:2' */
192     if (old_sf_car_DWork.is_active_gear_state != 0) {
193       switch (old_sf_car_DWork.is_gear_state) {
194        case old_sf_car_IN_first:
195         /* During 'first': '<S5>:6' */
```

Highlighted line of code

14 In the chart, right-click the transition with the condition `[speed > up_th]` and select **C/C++ Code > Navigate to C/C++ Code**.

The code for that transition appears highlighted in `old_sf_car.c`.



Note: For a list of the Stateflow objects in your model that are traceable, click the `Traceability Report` hyperlink in the code generation report.

Bidirectional Traceability for State Transition Tables

This example shows how to navigate bidirectionally between objects in a state transition table and the generated C/C++ and HDL code for traceability.

1 At the command prompt, type `sf_cdplayer_STT`. This model is already configured for traceability. For more information on these configurations, see "Traceability of Stateflow Objects in Generated Code" (Stateflow).

2 Press **Ctrl+B**.

This step generates source code and header files for the `sf_cdplayer_STT` model. After the code generation process is complete, the code generation report appears.

3 Click the `sf_cdplayer_STT.c` hyperlink in the report.

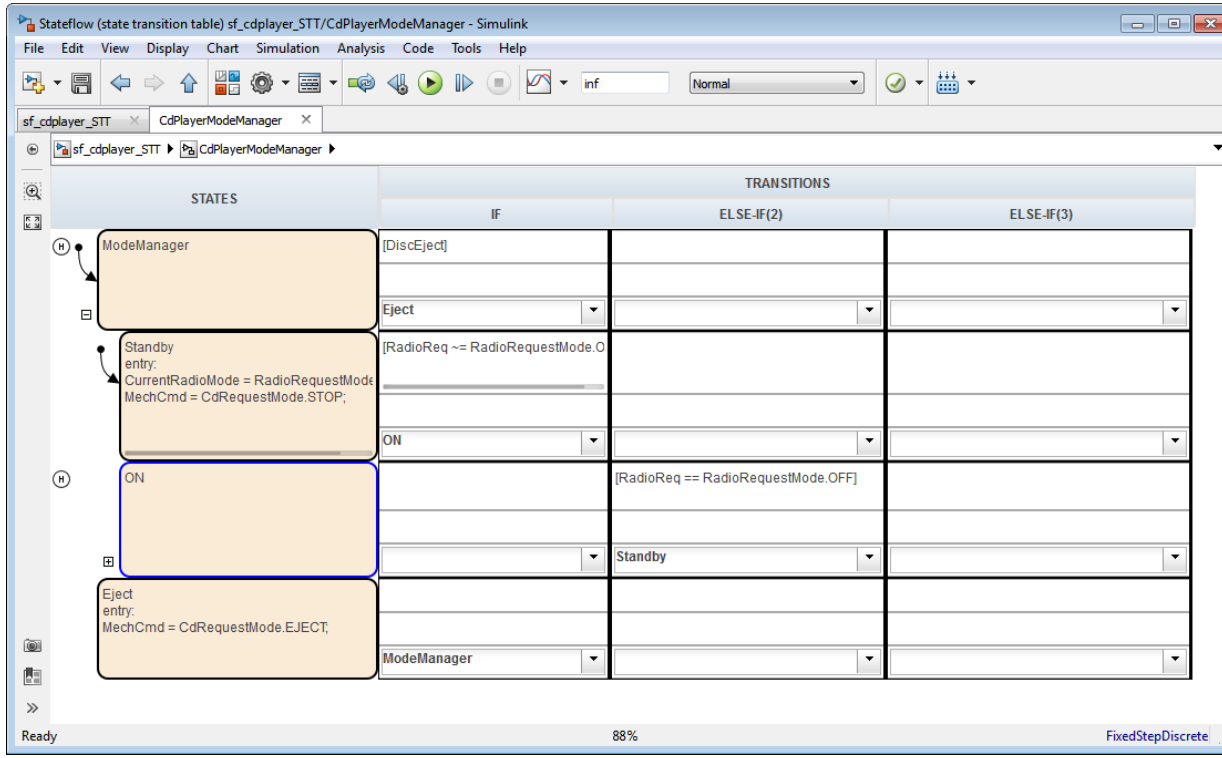
4 To see the traceability comments, scroll down through the code. The line numbers shown can differ from the numbers that appear in your code generation report.

```
60  /* Function for State Transition Table: '<Root>/CdPlayerModeManager' */
61  static void sf_cdplayer_S_enter_internal_ON(void)
62  {
63    /* Entry Internal 'ON': '<S2>:58' */
64    switch (sf_cdplayer_STT_DWork.bitsForTID1.was_ON) {
65     case sf_cdplayer_STT_IN_AMMode:
66      sf_cdplayer_STT_DWork.bitsForTID1.is_ON = sf_cdplayer_STT_IN_AMMode;
67      sf_cdplayer_STT_DWork.bitsForTID1.was_ON = sf_cdplayer_STT_IN_AMMode;
68
69      /* Entry 'AMMode': '<S2>:61' */
70      sf_cdplayer_STT_B.CurrentRadioMode = AM;
71      sf_cdplayer_STT_B.MechCmd = STOP;
72      break;
73
```

5 Click the `<S2>:58` hyperlink in this traceability comment:

```
/* Entry Internal 'ON': '<S2>:58' */
```

The corresponding state `'ON'` appears highlighted in the state transition table.



6 Right-click the highlighted state and select **View state object**. The state `'ON'` also appears highlighted in the underlying state transition diagram.

7 You can also trace a state or transition from the state transition table to the generated code. Right-click the state `Standby` and select **C/C++ Code > Navigate to C/C++ Code**.

The entry code for the state `Standby` is highlighted in the generated code.

```
234          sf_cdplayer_STT_DWork.bitsForTID1.was_ModeManager =
235            sf_cdplayer_STT_IN_Standby;
236
237          /* Entry 'Standby': '<S2>:57' */
238          sf_cdplayer_STT_B.CurrentRadioMode = OFF;
239          sf_cdplayer_STT_B.MechCmd = STOP;
240        } else if (sf_cdplayer_STT_DWork.RadioReq_prev !=
241                   sf_cdplayer_STT_DWork.RadioReq_start) {
242          /* Transition: '<S2>:75' */
243          if (sf_cdplayer_STT_P.RR_Value == CD) {
```

Bidirectional Traceability for Truth Table Blocks

See how bidirectional traceability works for a Truth Table block by following these steps:

1. At the command prompt, type `sf_climate_control`
2. Complete steps 2 through 5 in "Bidirectional Traceability for States and Transitions" on page 61-10.
3. To build the model, press **Ctrl+B**.

   The code generation report appears.
4. Click the `sf_climate_control.c` hyperlink in the report.
5. To see the traceability comments, scroll down through the code. The following line numbers can differ from the numbers that appear in your code.

```
77       /*  Turn On Humidifier */
78       /* Action '3': '<S1>:1:47' */
79       rtb_humidifier = 1;
80     } else if (eml_aVarTruthTableCondition) {
81       /* Decision 'D2': '<S1>:1:18' */
```

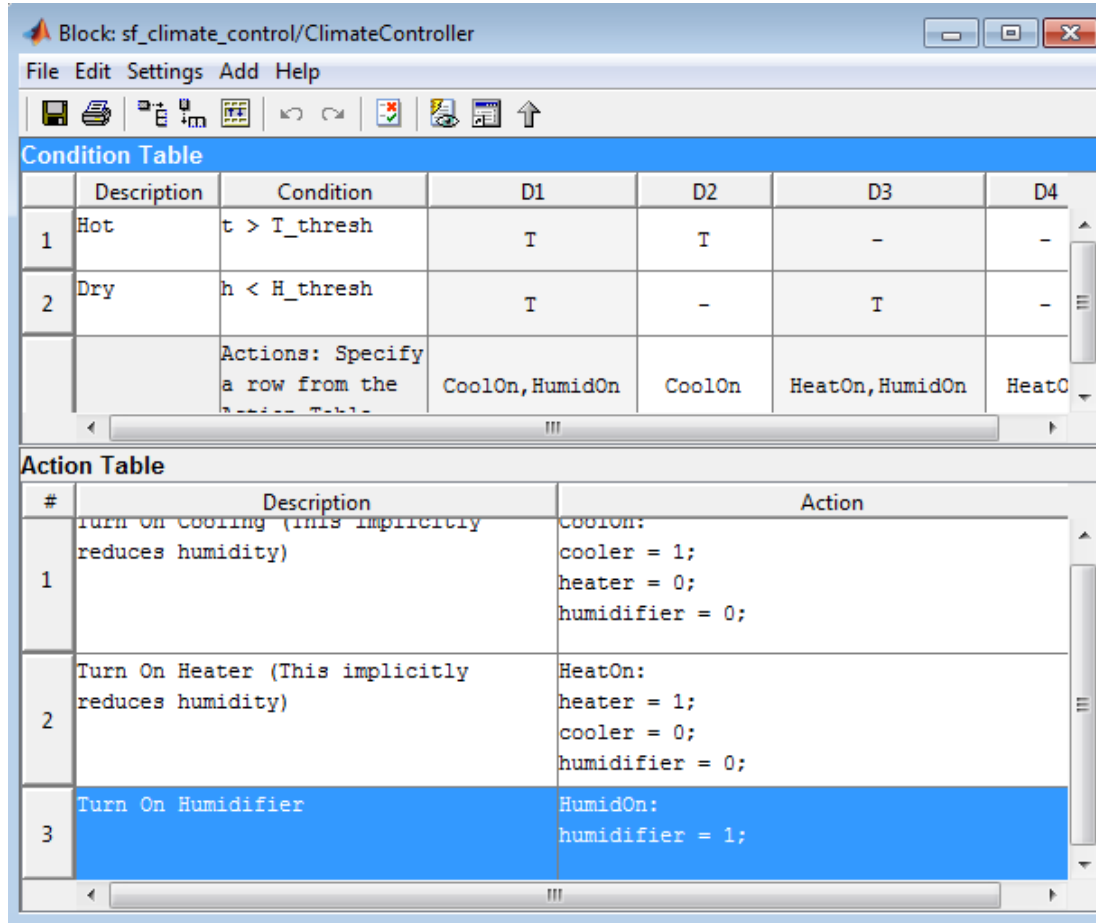
Traceability comment for a truth table action

Traceability comment for a truth table decision

6. Click the `<S1>:1:47` hyperlink in this traceability comment:

   ```
   /* Action '3': '<S1>:1:47' */
   ```

   In the Truth Table Editor, row 3 of the Action Table appears highlighted.



7 You can also trace a condition, decision, or action in the table to a line of generated code. For example, right-click a cell in the column `D2` and select **C/C++ Code** > **Navigate to C/C++ Code**.

The code for that decision appears highlighted in `sf_climate_control.c`.

```
77        /*  Turn On Humidifier */
78        /* Action '3': '<S1>:1:47' */
79        rtb_humidifier = 1;
80      } else if (eml_aVarTruthTableCondition) {
81        /* Decision 'D2': '<S1>:1:18' */
```

Highlighted line of code

Tip: To select **C/C++ Code** > **Navigate to C/C++ Code** for a condition, decision, or action, right-click a cell in the row or column that corresponds to that truth table element.

Bidirectional Traceability for Graphical Functions

See how bidirectional traceability works for graphical functions by following these steps:

1. At the command prompt, type `sf_clutch`.
2. Complete steps 2 through 6 in "Bidirectional Traceability for States and Transitions" on page 61-10.
3. In the Model Configuration Parameters dialog box, go to the **Solver** pane. In the **Solver options** section, select `Fixed-step` in the **Type** field. Click **Apply**. Before generating code, you must perform this step because the model does not work with variable step solvers.
4. To build the model, press **Ctrl+B**.

   The code generation report appears.
5. Click the `sf_clutch.c` hyperlink in the report.
6. To see the traceability comments, scroll down through the code. The following line numbers can differ from the numbers that appear in your code generation report.

```
235           case sf_clutch_IN_Slipping:
236             /* Graphical Function 'detectLockup': '<S1>:10' */
237             /* Transition: '<S1>:28' */
238             /* Graphical Function 'getSlipTorque': '<S1>:3' */
```

Traceability comment for a graphical function

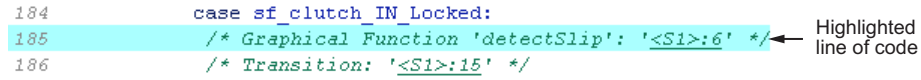
7. Click the `<S1>:3` hyperlink in this traceability comment:

   ```
   /* Graphical Function 'getSlipTorque': '<S1>:3' */
   ```

   In the chart, the graphical function `getSlipTorque` appears highlighted.

8 You can also trace a graphical function in the chart to a line of generated code. For example, right-click the graphical function `detectSlip` and select **C/C++ Code** > **Navigate to C/C++ Code**.

The code for that graphical function appears highlighted in `sf_clutch.c`.



Code-to-Model Traceability for Events

See how code-to-model traceability works for events by following these steps:

1 At the command prompt, type `sf_security`.

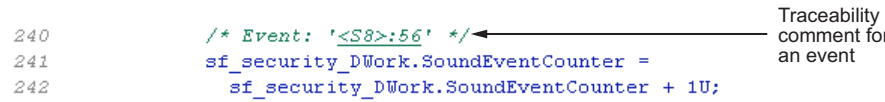
2 Complete steps 2 through 6 in "Bidirectional Traceability for States and Transitions" on page 61-10.

3 To build the model, press **Ctrl+B**.

The code generation report appears.

4 Click the `sf_security.c` hyperlink in the report.

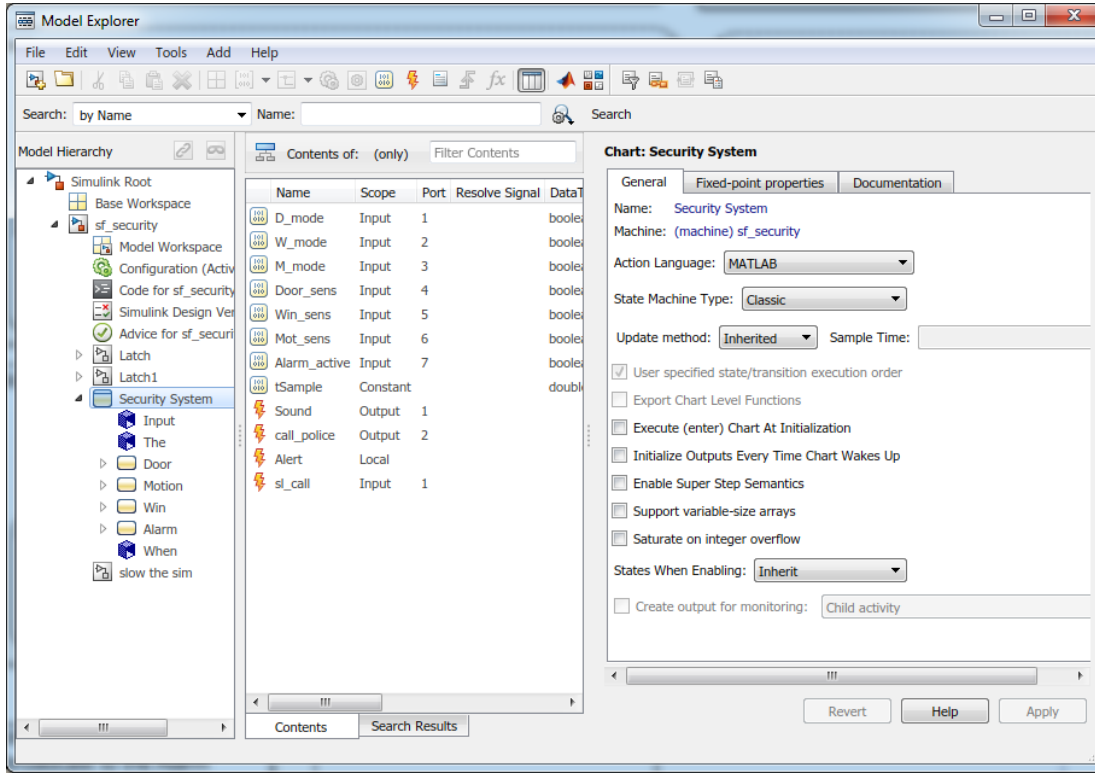
5 To see the following traceability comment, scroll down through the code. The following line numbers can differ from the numbers that appear in your code generation report.



6 Click the `<S8>:56` hyperlink in this traceability comment:

```
/* Event: '<S8>:56' */
```

In the **Contents** pane of the Model Explorer, the event `Sound` appears highlighted.



Model-to-Code Traceability for Junctions

See how model-to-code traceability works for junctions by following these steps:

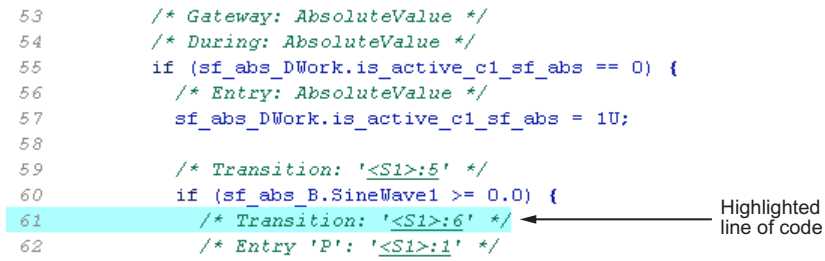
1. At the command prompt, type `sf_abs`.
2. Complete steps 2 through 6 in "Bidirectional Traceability for States and Transitions" on page 61-10.
3. In the Model Configuration Parameters dialog box, go to the **Solver** pane. In the **Solver options** section, select `Fixed-step` in the **Type** field. Click **Apply**. Before generating code, you must perform this step because the model does not work with variable-step solvers.
4. To build the model, press **Ctrl+B**.

The code generation report appears.

5 Open the `AbsoluteValue` chart.

6 Right-click the left junction and select **C/C++ Code** > **Navigate to C/C++ Code**.

The code for the first outgoing transition of that junction appears highlighted in `sf_abs.c`.



Format of Traceability Comments for Stateflow Objects

The format of a traceability comment depends on the Stateflow object type.

State

Syntax

```
/* <ActionType> '<StateName>': '<ObjectHyperlink>' */
```

Example

```
/* During 'gear_state': '<S5>:2' */
```

This comment refers to the `during` action of the state `gear_state`, which has the hyperlink `<S5>:2`.

Transition

Syntax

```
/* Transition: '<ObjectHyperlink>' */
```

Example

```
/* Transition: '<S5>:12' */
```

This comment refers to a transition, which has the hyperlink `<S5>:12`.

MATLAB Function

Syntax

```
/* MATLAB Function '<Name>': '<ObjectHyperlink>' */
```

Within the inlined code for a MATLAB function, comments that link to individual lines of the function have the following syntax:

```
/* '<ObjectHyperlink>' */
```

Examples

```
/* MATLAB Function 'test_function': '<S50>:99' */

/* '<S50>:99:20' */
```

The first comment refers to the MATLAB function named `test_function`, which has the hyperlink `<S50>:99`.

The second comment refers to line 20 of the MATLAB function in your chart.

Truth Table Block

Syntax

```
/* Truth Table Function '<Name>': '<ObjectHyperlink>' */
```

Within the inlined code for a Truth Table block, comments for conditions, decisions, and actions have the following syntax:

```
/* Condition '#<Num>': '<ObjectHyperlink>' */
/* Decision 'D<Num>': '<ObjectHyperlink>' */
/* Action '<Num>': '<ObjectHyperlink>' */
```

`<Num>` is the row or column number that appears in the Truth Table Editor.

Examples

```
/* Truth Table Function 'truth_table_default': '<S10>:100' */

/* Condition '#1': '<S10>:100:8' */
/* Decision 'D1': '<S10>:100:16' */
/* Action '1': '<S10>:100:31' */
```

The first comment refers to a Truth Table block named `truth_table_default`, which has the hyperlink `<S10>:100`.

The other three comments refer to elements of that Truth Table block. Each condition, decision, and action in the Truth Table block has a unique hyperlink.

Truth Table Function

For syntax and examples, see "Truth Table Block" on page 61-21.

Graphical Function

Syntax

```
/* Graphical Function '<Name>': '<ObjectHyperlink>' */
```

Example

```
/* Graphical Function 'hello': '<S1>:123' */
```

This comment refers to a graphical function named `hello`, which has the hyperlink `<S1>:123`.

Simulink Function

Syntax

```
/* Simulink Function '<Name>': '<ObjectHyperlink>' */
```

Example

```
/* Simulink Function 'simfcn': '<S4>:10' */
```

This comment refers to a Simulink function named `simfcn`, which has the hyperlink `<S4>:10`.

Event

Syntax

```
/* Event: '<ObjectHyperlink>' */
```

Example

```
/* Event: '<S3>:33' */
```

This comment refers to an event, which has the hyperlink `<S3>:33`.

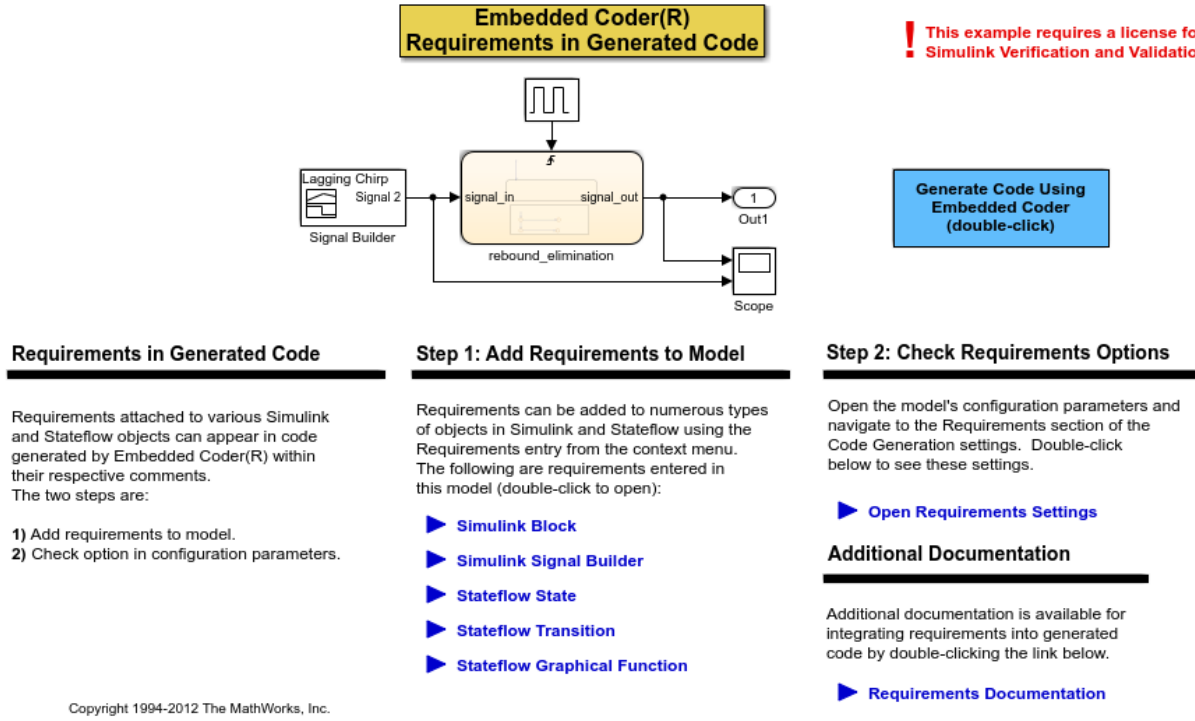
Link Generated Code to Requirements

Link generated code to model object requirements. Using configuration parameters, you can specify whether to include requirement descriptions as comments in the generated code.

Open Model

Open the rtwdemo_requirements model. The model contains Simulink® and Stateflow® objects with associated requirements.

```
model='rtwdemo_requirements';
open_system(model);
```



View Requirements

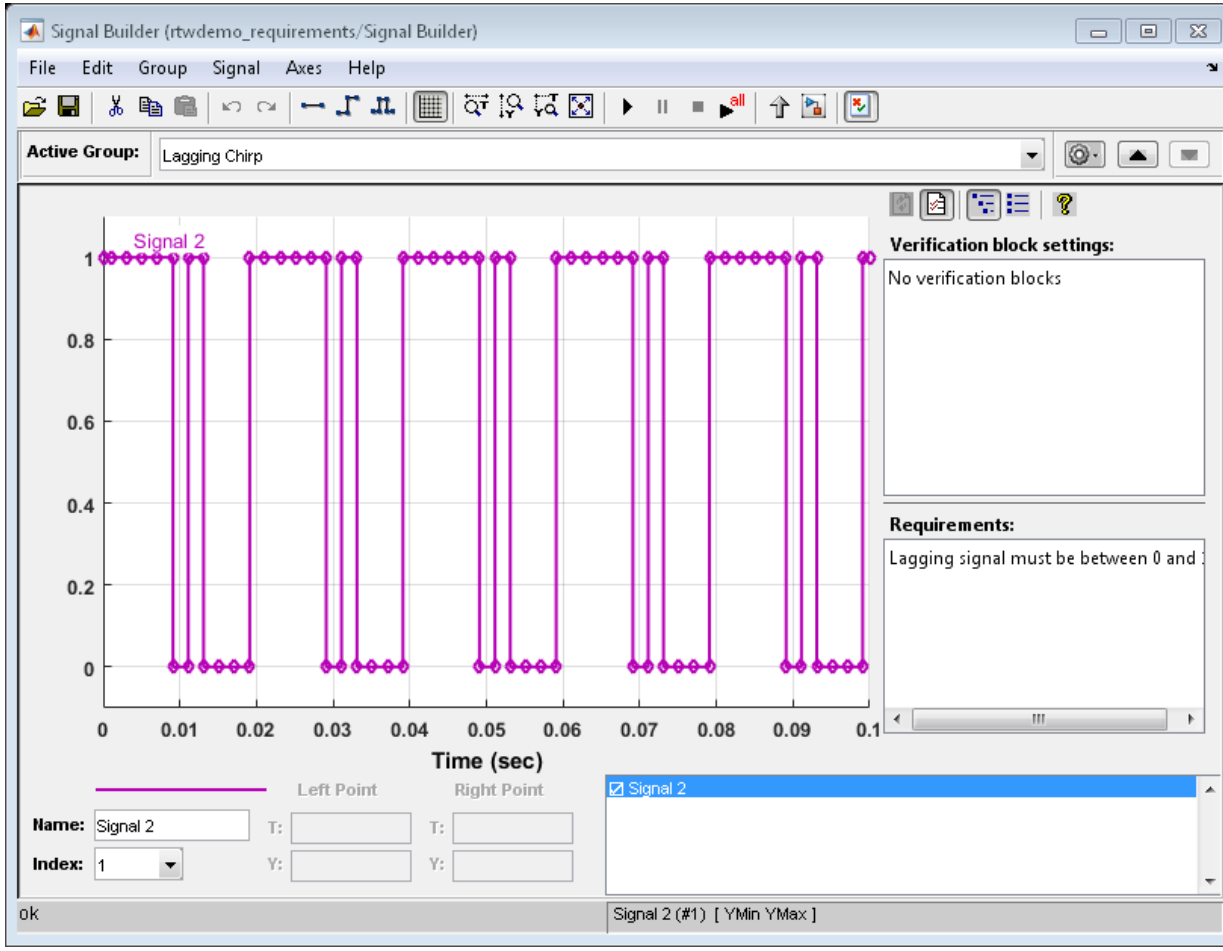
You can view requirements to model objects by using the object context menu. Right-click an object and select **Requirements Traceability**. To view the requirements, use these commands:

1. To view the requirements for the `DiscretePulseGenerator` block, open the Link Editor.

```
clockblock='rtwdemo_requirements/clock';
clockblockh=get_param(clockblock,'handle');
rmi('edit',clockblockh);
```

2. To view the requirements, open the **Signal Builder** block.

```
sigbblock='rtwdemo_requirements/Signal Builder';
open_system(sigbblock)
```



3. To view the requirements for the Stateflow® state, open the Link Editor.

```
state=find(sfroot,'-isa','Stateflow.State','-and','Tag','req_state');
rmi('edit',state.id);
```

4. To view the requirements for the Stateflow transition, open the Link Editor.

```
trans=find(sfroot,'-isa','Stateflow.Transition','-and','Tag','req_trans');
rmi('edit',trans.id);
```

5. To view the requirements for the Stateflow function, open the Link Editor.

```
func=find(sfroot,'-isa','Stateflow.Function','-and','Tag','req_function');
rmi('edit',func.id);
```

Close the open windows.

```
close_system(sigbblock);
```

Set Configuration Parameters

Open the Configuration Parameters dialog box **Code Generation > Comments** pane. View the configuration parameter settings.

```
model = bdroot;
slCfgPrmDlg(model,'Open');
slCfgPrmDlg(bdroot,'TurnToPage','Comments');
```

Generate Code

Generate code for the model.

```
rtwbuild('rtwdemo_requirements')
```

```
### Starting build procedure for model: rtwdemo_requirements
### Successful completion of build procedure for model: rtwdemo_requirements
```

In the generated code, view the comments containing the requirements.

```
rtwdemodbtype('rtwdemo_requirements_ert_rtw/rtwdemo_requirements.c',...
    '/* Function for Chart:','return result;',1,0);
```

```
/* Function for Chart: '<Root>/rebound_elimination' */
static real_T rebound_fcn(real_T prev_in, real_T prev_out, real_T curr_in)
{
  real_T result;

  /* Graphical Function 'rebound_fcn': '<S2>:2':
   *  1. Result Computation
   */
  /* Transition: '<S2>:4' */
  if (prev_in == curr_in) {
    /* Transition: '<S2>:5' */
    result = curr_in;
  } else {
    /* Transition: '<S2>:6' */
    /* Transition: '<S2>:7' */
```

```
    result = prev_out;
  }
```

See Also

- For requirement traceability, see Overview of the Requirements Management Interface

Close Model

```
rtwdemoclean;
close_system('rtwdemo_requirements',0);
```

Reload Existing Traceability Information

To reload existing traceability information for a model:

1. In the Configuration Parameters dialog box, on the **All Parameters** tab, under Model-to-code, click **Configure**.
2. In the Model-to-code navigation dialog box, in the **Build folder** field, type or browse to the build folder that contains the existing traceability information.

If you close and reopen a model, the **Navigate to Code** context menu option might not be available because Embedded Coder cannot find a build folder for your model in the current working folder. Without having to reset the current working folder or rebuild the model, do the following:

1. To open the Model-to-code navigation dialog box, click **Configure**.
2. In the Model-to-code navigation dialog box, click **Browse**.
3. Browse to the build folder for your model, and select the folder. The build folder path is displayed in the **Build folder** field.
4. If you selected **Model-to-code** for the build, clicking **Apply** or **OK** loads traceability information from the earlier build into your Simulink session.
5. To open the context menu and trace a model object to corresponding code, right-click a model object and select **C/C++ Code** > **Navigate to C/C++ Code**.

Customize Traceability Reports

In the Configuration Parameters dialog box, the **Code Generation** section lists parameters that you can select and clear to customize the content of your traceability reports.

Select or clear any combination of the following parameters, which are on by default:

- **Eliminated / virtual blocks** (Simulink Coder) (account for blocks that are untraceable)
- **Traceable Simulink blocks** (Simulink Coder)
- **Traceable Stateflow objects** (Simulink Coder)
- **Traceable MATLAB functions** (Simulink Coder)

If you select all parameters, you get a complete mapping between model elements and the generated code.

The following figure shows the top section of the traceability report that is generated when you select all traceability content parameters for model `rtwdemo_hyperlinks`.

Traceability Report for rtwdemo_hyperlinks

Generate Traceability Matrix

Table of Contents

Eliminated / Virtual Blocks

Block Name	Comment
<Root>/Build ERT	Empty SubSystem
<Root>/Mux	Mux
<Root>/Scope	Unused code path elimination
<Root>/View Code Generation Report	Empty SubSystem

Traceable Simulink Blocks / Stateflow Objects / MATLAB Functions

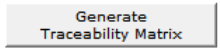
Root system: rtwdemo_hyperlinks

Object Name	Code Location
<Root>/Chart	rtwdemo_hyperlinks.c:20, 43, 85, 103, 112, 143, 257 rtwdemo_hyperlinks.h:39, 40, 41, 43, 45, 46, 47, 48, 49, 52, 53
<Root>/Constant	rtwdemo_hyperlinks.c:144

Generate a Traceability Matrix

If you have DO Qualification Kit software or IEC Certification Kit software and are using a Windows host, you can generate a traceability matrix into Microsoft Excel® format directly from the traceability report. See “Customize Traceability Reports” on page 61-29.

Go to the **Traceability Report** section of the HTML code generation report and click **Generate Traceability Matrix**.



To select an existing matrix file to update or specify a new matrix file to create, use the options in the Generate Traceability Matrix dialog box. Optionally, you can select and order the columns that appear in the generated matrix. For more information, see “Generating a Traceability Matrix” in either the DO Qualification Kit documentation (DO Qualification Kit) or the IEC Certification Kit documentation (IEC Certification Kit).

Traceability Limitations

These limitations apply to reports generated by Embedded Coder software.

- Under the following conditions, model-to-code traceability is disabled for a block if the block name contains:
  - A single quote (').
  - An asterisk (*), that causes a name-mangling ambiguity relative to other names in the model. This name-mangling ambiguity occurs if in a block name or at the end of a block name, an asterisk precedes or follows a slash (/).
  - The character ÿ (`char(255)`).
- If a block name contains a newline character (`\n`), in the generated code comments, the block path name hyperlink replaces the newline character with a space for readability.
- You cannot trace blocks representing these types of subsystems to generated code:
  - Virtual subsystems
  - Masked subsystems
  - Nonvirtual subsystems for which code has been optimized away

  If you cannot trace a subsystem at subsystem level, you might be able to trace individual blocks within the subsystem.
- If you open a model on a platform that is different from the platform used to generate code, you cannot use model-to-code and code-to-model traceability features.

Component Verification in Embedded Coder

Component Verification in the Target Environment

After you generate production code for a component design, you need to integrate, compile, link, and deploy the code as a complete application on the embedded system. One approach is to manually integrate the code into an existing software framework that consists of an operating system, device drivers, and support utilities. The algorithm can include externally written legacy or custom code.

An easier approach to verifying a component in a target environment is to use processor-in-the-loop (PIL) simulation. For information about PIL simulations, see “SIL and PIL Simulations” on page 64-2.

Goals of Component Verification

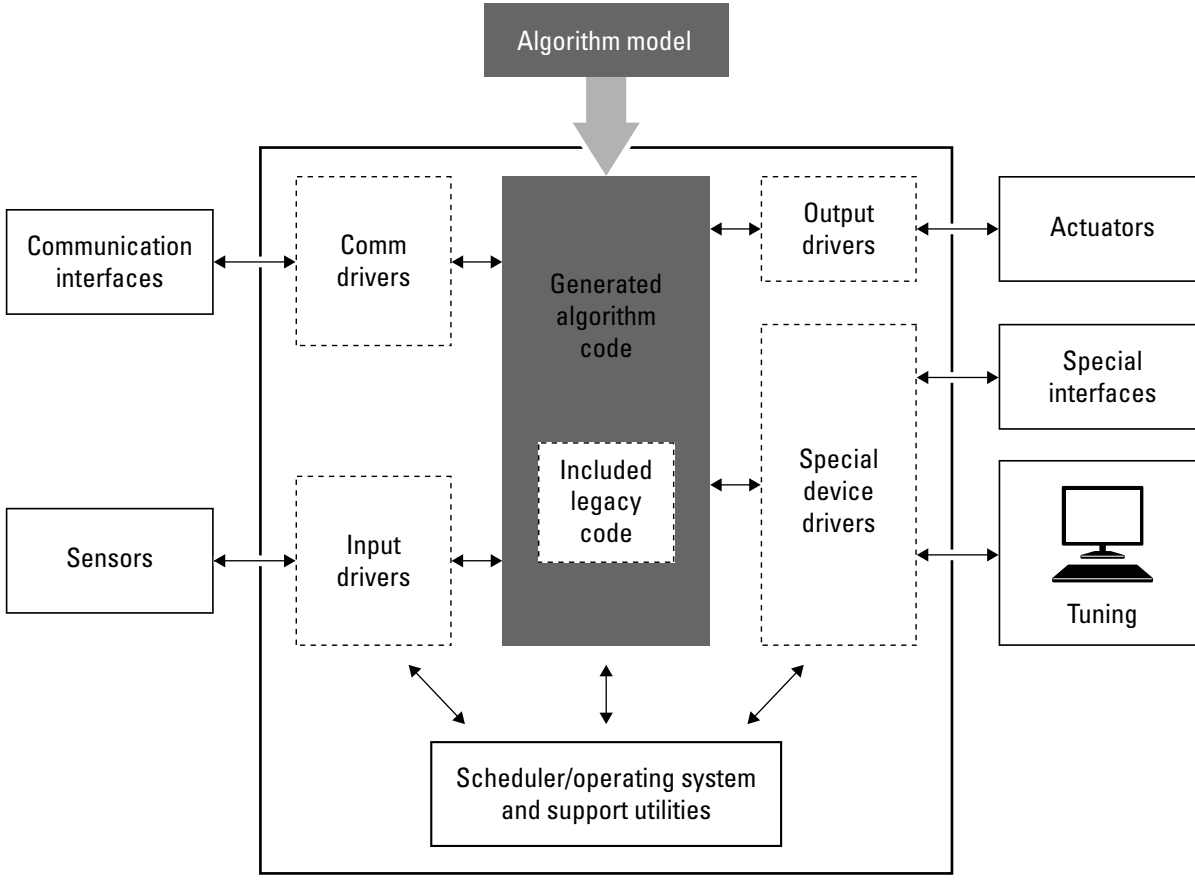
Assuming that you have generated production source code and integrated required externally written code, such as drivers and a scheduler, you can verify that the integrated software operates as expected by testing it in the target environment. During testing, you can achieve either of the following goals, depending on whether you export code that is strictly ANSI C/C++ or mixes ANSI C/C++ with code optimized for a target environment.

Goal	Type of Code Export
Maximize code portability and configurability	ANSI C/C++
Simplify integration and maximize use of processor resources and code efficiency	Mixed code

Regardless of your goal, you must integrate required external drivers and scheduling software. To achieve real-time execution, you must integrate the real-time scheduling software.

Maximizing Code Portability and Configurability

To maximize code portability and configurability, limit the application code to ANSI/ISO C or C++ code only, as the following figure shows.

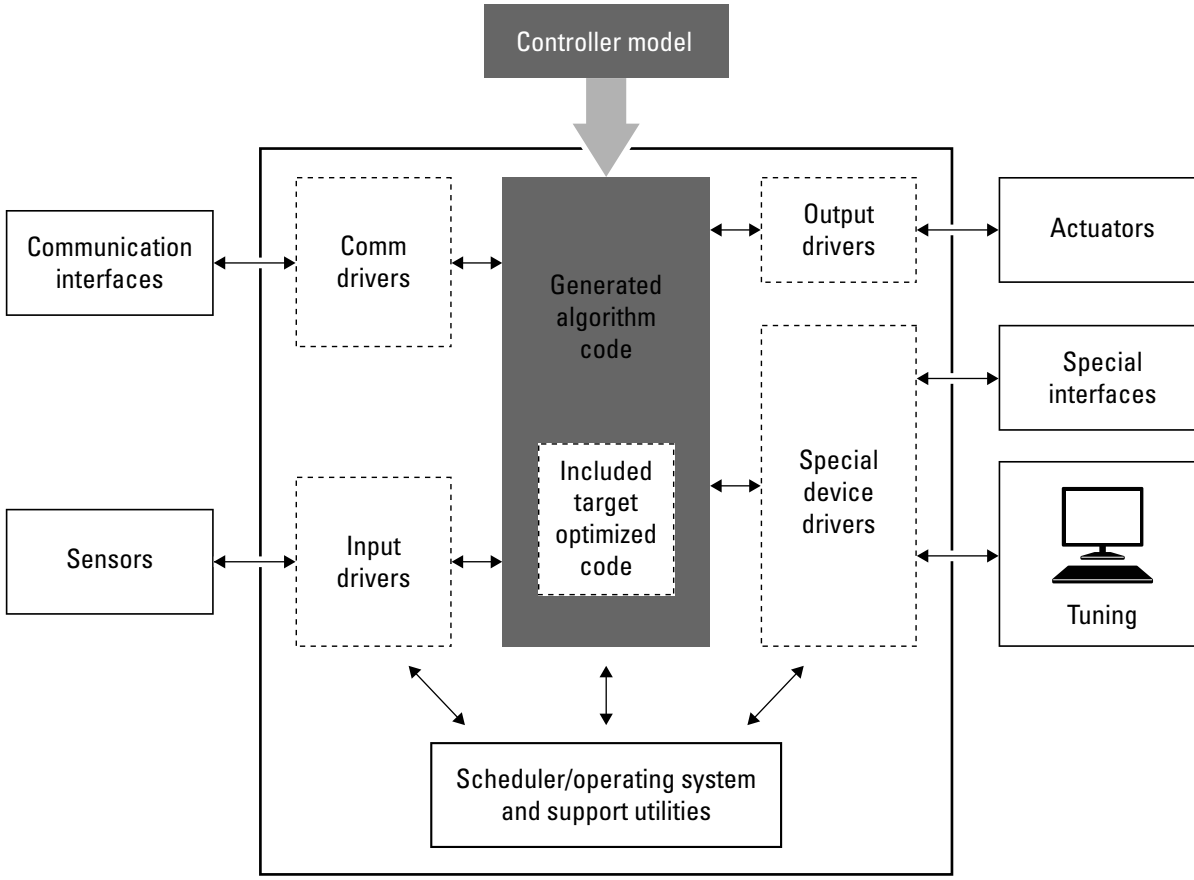


Simplifying Code Integration and Maximizing Code Efficiency

To simplify code integration and maximize code efficiency for a target environment, use Embedded Coder features for:

- Controlling code interfaces
- Exporting subsystems
- Including target-specific code, including compiler optimizations

The following figure shows a mix of ANSI C/C++ code with code that is optimized for a target environment.



Running Component Tests

The workflow for running software component tests in the target environment is:

1. Integrate external code, for example, for device drivers and a scheduler, with the generated C or C++ code for your component model. For more information, see "S-Functions and Code Generation" (Simulink Coder) in the Simulink Coder documentation. For more specific references that depend on your verification goals, see the following table.

For	See
ANSI C/C++ code integration	"Integrate C Functions Using Legacy Code Tool" (Simulink) in the Simulink documentation. Also, open `rtwdemos` and navigate to the **Custom Code** folder.
Mixed code integration	• "Generate Component Source Code for Export to External Code Base" on page 39-51 and example `rtwdemo_exporting_functions` • "Control Generation of Function Prototypes" on page 26-2, "Control Generation of C++ Class Interfaces" on page 26-23, and example rtwdemo_fcnprotoctrl • "What Is Code Replacement?" on page 38-2, "What Is Code Replacement Customization?" on page 51-3, and example `rtwdemo_crl_script`

2. Simulate the integrated component model.
3. Generate code for the integrated component model.
4. Connect to data interfaces for the generated C code data structures. See "Exchange Data Between Generated and External Code Using C API" (Simulink Coder) and "Export ASAP2 File for Data Measurement and Calibration" (Simulink Coder) in the Simulink Coder documentation. Also see examples `rtwdemo_capi` and `rtwdemo_asap2`.
5. Customize and control the build process, if required. See "Customize Post-Code-Generation Build Processing" (Simulink Coder) in the Simulink Coder documentation, and example `rtwdemo_buildinfo` .

6 Create a zip file that contains generated code files, static files, and dependent data to build the generated code in an environment other than your host computer. See "Relocate Code to Another Development Environment" (Simulink Coder), in the Simulink Coder documentation, and example `rtwdemo_buildinfo`.

Component Verification With a Real-Time Target Environment in Embedded Coder

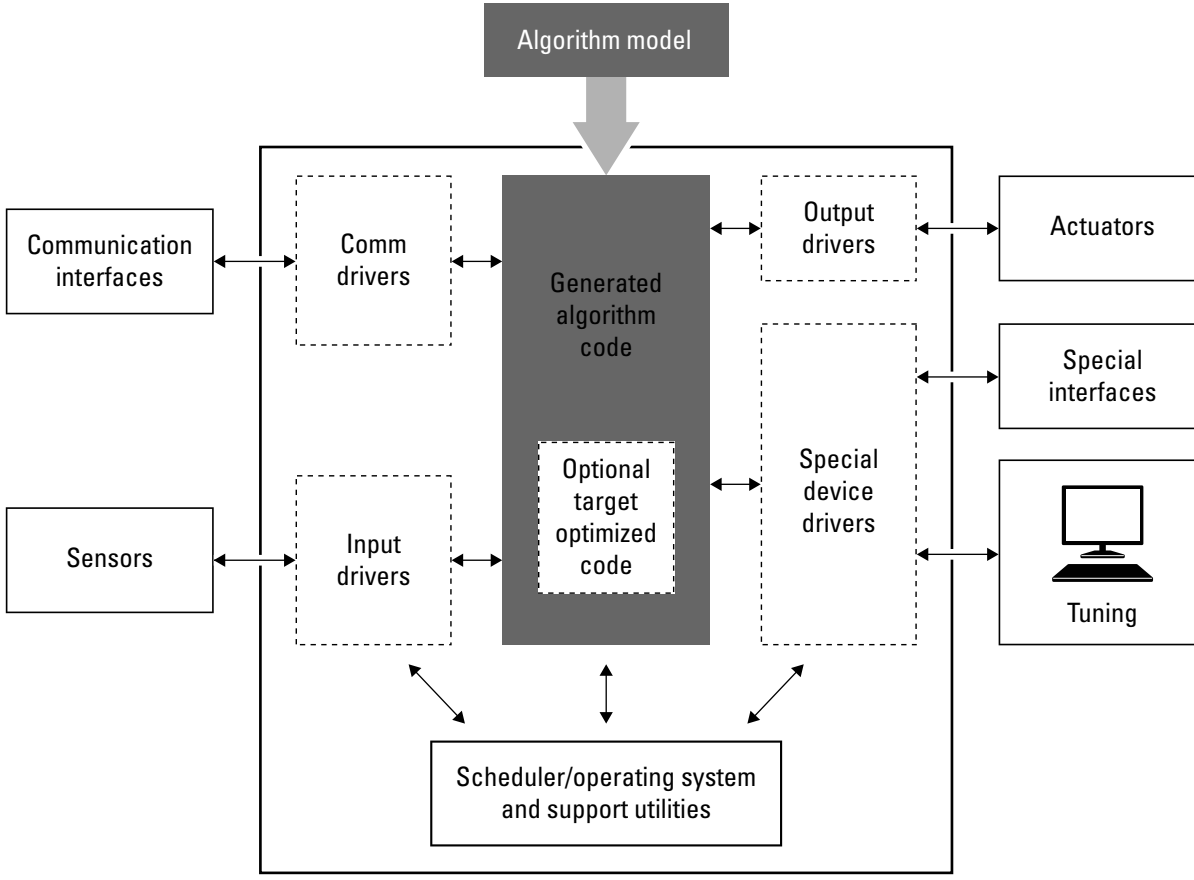
About Real-Time Software Component Verification

One approach to verifying a software component is to build the component into a complete software system that can execute in real time in the target environment. A complete software system includes:

- Algorithm for the software component
- Scheduling algorithms
- Calls to drivers for board-specific devices

This single build approach is more time consuming to set up, but makes it easier to get the complete application running in the target environment.

The following figure shows code generated for an algorithm being built into a complete system executable for the target environment.


Algorithm model
Communication interfaces
Comm drivers
Generated algorithm code
Output drivers
Actuators
Special interfaces
Optional target optimized code
Special device drivers
Sensors
Input drivers
Tuning
Scheduler/operating system and support utilities

Real-Time Software Component Testing

The workflow for testing component software as part of a complete real-time target environment is:

1. Develop a component model and generate source code for production.

   For information on building in scheduling and real-time system support, see:

   - "Time-Based Scheduling and Code Generation" (Simulink Coder) and "Modeling for Multitasking Execution" (Simulink Coder) in the Simulink Coder documentation. For an example, open `rtwdemos` and navigate to the **Multirate Support** folder.
   - "Asynchronous Events" (Simulink Coder) in the Simulink Coder documentation and example `rtwdemo_async`
   - "Deploy Generated Standalone Executable Programs To Target Hardware" on page 49-2
   - "Workflows for AUTOSAR" and example "Generate AUTOSAR-Compliant C Code and Export ARXML Descriptions".
2. Optimize generated code for a specific run-time environment, using specialized function libraries. For more information, see "What Is Code Replacement?" on page 38-2, "What Is Code Replacement Customization?" on page 51-3, and "Optimize Generated Code By Developing and Using Code Replacement Libraries - Simulink®".
3. Customize post code generation build processing to accommodate third-party tools and processes, as required. See "Customize Post-Code-Generation Build Processing" (Simulink Coder) in the Simulink Coder documentation and example `rtwdemo_buildinfo`.
4. Integrate external code, for example, for device drivers and a scheduler, with the generated C or C++ code for your component model. For more information, see "S-Functions and Code Generation" (Simulink Coder) in the Simulink Coder documentation. For more specific references depending on your verification goals, see the following table.

For...	See...
ANSI C/C++ code integration	"Integrate C Functions Using Legacy Code Tool" (Simulink) in the Simulink documentation. Also, open `rtwdemos` and navigate to the **Custom Code** folder.

For...	See...
Mixed code integration	• "Generate Component Source Code for Export to External Code Base" on page 39-51 and example `rtwdemo_exporting_functions` • "Control Generation of Function Prototypes" on page 26-2, "Control Generation of C++ Class Interfaces" on page 26-23, and example rtwdemo_fcnprotoctrl • "What Is Code Replacement?" on page 38-2, "What Is Code Replacement Customization?" on page 51-3, and example "Optimize Generated Code By Developing and Using Code Replacement Libraries - Simulink®"

5 Simulate the integrated model.

6 Generate code for the integrated model.

7 Connect to data interfaces for the generated C code data structures. See "Exchange Data Between Generated and External Code Using C API" (Simulink Coder) and "Export ASAP2 File for Data Measurement and Calibration" (Simulink Coder) in the Simulink Coder documentation. Also see examples `rtwdemo_capi` and `rtwdemo_asap2`.

8 Customize and control the build process, as required. See "Customize Post-Code-Generation Build Processing" (Simulink Coder), in the Simulink Coder documentation, and example `rtwdemo_buildinfo` .

9 Create a zip file that contains generated code files, static files, and dependent data to build the generated code in an environment other than your host computer. See "Relocate Code to Another Development Environment" (Simulink Coder), in the Simulink Coder documentation, and example `rtwdemo_buildinfo`.

Numerical Equivalence Checking in Embedded Coder

SIL and PIL Simulations

In this section...

What Are SIL and PIL Simulations?

With Embedded Coder, you can run software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulations of your model. These simulations generate source code for either the top model or part of the model. A SIL simulation compiles and runs the generated code on your development computer. A PIL simulation cross-compiles source code on your development computer, and then downloads and runs the object code on a target processor or an equivalent instruction set simulator.

With SIL and PIL simulations, you can:

- Test whether your model and generated code are numerically equivalent.
- Observe code coverage.
- Perform code execution profiling.

Why Use SIL and PIL

Through SIL and PIL, you can early on test and fix defects. For example, you can model and test a system component in normal mode. Then, you can reuse your test suites in a SIL or PIL simulation that runs compiled generated code. To check numerical equivalence, you compare normal and SIL or PIL simulation results. You thereby avoid leaving the Simulink environment to test generated code on a separate infrastructure.

This table describes situations where you can use SIL and PIL.

Situation	Use
Reuse test vectors developed for normal mode simulation to verify numerical output of generated (or legacy) code. For example, reusing test cases generated by Simulink Design Verifier™. See "What Is Test Case Generation?" (Simulink Design Verifier) in Simulink Design Verifier documentation.	SIL and PIL
Collect metrics for generated code: • Code coverage. See "Configure Code Coverage with Third-Party Tools" on page 67-10. • Execution profiling. See "Code Execution Profiling with SIL and PIL" on page 58-2 • Stack profiling. See "Perform Stack Profiling with IDE and Toolchain Targets" on page 73-22.	SIL and PIL
Achieve IEC 61508, IEC 62304, ISO 26262, or DO-178 certification. See "Embedded Coder Reference Workflow Overview" (IEC Certification Kit) in the IEC Certification Kit documentation and Testing of Outputs of Integration Process (DO Qualification Kit) in the DO Qualification Kit documentation.	SIL and PIL
Without target hardware, get a convenient alternative to PIL.	SIL

Situation	Use
With target hardware, for example, an evaluation board or instruction set simulator: • Verify behavior of target-specific code, for example, code replacement optimizations, and legacy code. See “What Is Code Replacement?” on page 38-2 and “What Is Code Replacement Customization?” on page 51-3. • Optimize the execution speed and memory footprint of your code. In this table, see the information about collecting execution profiling and stack profiling metrics. • Investigate effects of compiler settings and optimizations, for example, deviation from ANSI C overflow behavior. Normal simulation techniques do not account for restrictions and requirements that the hardware imposes, such as limited memory resources or behavior of target-specific optimized code. For information about running PIL simulations on specific targets, see “Sample Custom Targets” (Simulink Coder) in the Simulink Coder documentation.	PIL

Note: The SIL and PIL simulation modes are not designed for the reduction of model simulation times. If you want to speed up the simulation of your model, use the rapid accelerator mode. For more information, see “What Is Acceleration?” (Simulink).

How SIL and PIL Simulations Work

In a SIL or PIL simulation, code is generated for either the top model or part of the model. With SIL, this code is compiled for and executed on your development computer. With PIL, the code is cross-compiled for the target hardware and runs on the target processor.

Through a communication channel, Simulink sends stimulus signals to the code on your computer or target processor for each sample interval of the simulation.

- For a top model, Simulink uses stimulus signals from the base or model workspace.

- If you have designated only part of the model to simulate in SIL or PIL mode, then a part of the model remains in Simulink and code is not generated for this part of the model. Typically, you configure this part of the model to provide test vectors for the software executing on the hardware. This part of the model can represent other parts of the algorithm or the environment in which the algorithm operates.

When your computer or target processor receives signals from Simulink, the processor executes the SIL or PIL algorithm for one sample step. The SIL or PIL algorithm returns output signals calculated during this step to Simulink through a communication channel. One sample cycle of the simulation is complete, and Simulink proceeds to the next sample interval. The process keeps repeating itself and the simulation progresses. SIL and PIL simulations do not run in real time. In each sample period, Simulink and the object code exchange I/O data.

Comparison of SIL and PIL Simulations

Type of SIL or PIL Simulation	What Happens in SIL Simulation	What Happens in PIL Simulation
Specify through: • Top-model simulation mode • Model block **Simulation mode** parameter	• Test behavior of generated source code on development computer. Simulation does not test code compiled for target hardware because code is compiled for the development computer (different compiler and different processor architecture than the target). • Generated production code compiled and executed on development computer as separate process, independent of MATLAB process. • Execution is host/host and nonreal time.	• Test object code that you intend to deploy in production on either real target hardware or an instruction set simulator. • On development computer, generated production code cross-compiled for target. Object code downloaded and executed on target processor or instruction set simulator. • Execution is host/target and nonreal time.

Type of SIL or PIL Simulation	What Happens in SIL Simulation	What Happens in PIL Simulation
Use SIL or PIL block created from subsystem.	• Simulation runs compiled object code through S-function. S-function communicates with object code executing as standalone application on development computer. SIL block execution is independent of the MATLAB process. • Execution is host/host and nonreal time.	• Simulation runs cross-compiled object code through S-function on development computer. S-function communicates with object code executing as standalone application on target processor or instruction set simulator. • Execution is host/target and nonreal time.

Code Interfaces for SIL and PIL

You generate standalone code when you perform, for example, a top-model or right-click subsystem build for a single deployable component. You can compile and link standalone code into a standalone executable or integrate it with other code. For more information on the standalone code interface, see "Entry-Point Functions and Scheduling" (Simulink Coder).

When you generate code for a referenced model hierarchy, the software generates standalone executable code for the top model and a library module called a *model reference target* for each referenced model. When the code executes, the standalone executable invokes the applicable model reference targets to compute the referenced model outputs. For more information, see "Build Model Reference Targets" (Simulink Coder).

To integrate generated code with legacy code, use standalone code because the standalone code interface is documented.

Note: SIL and PIL simulations do not provide direct support for custom code interfaces. You can incorporate these interfaces into Simulink as an S-function, for example, using the Legacy Code Tool, S-Function Builder, or handwritten code. Then, you can verify the custom code by using SIL and PIL simulations.

This table provides the interfaces that SIL and PIL simulations generate.

SIL/PIL Simulation	Code Interface
Top-model	SIL/PIL simulation generates the standalone code interface. If code exists, simulation calls standalone code for the model . If code does not exist, simulation generates standalone code.
Model block	If you set **Code interface** block parameter to `Top model`, SIL/PIL simulation generates standalone code interface. Simulation calls standalone code for the model if it exists. Otherwise, simulation generates standalone code by using `slbuild('model')` command. If you set **Code interface** block parameter to `Model reference`, SIL/PIL simulation generates model reference code interface. Simulation calls model reference target for Model block if it exists. Otherwise, simulation generates model reference target by using `slbuild('model', 'ModelReferenceRTWTarget')` command.
SIL or PIL block	Block uses standalone code interface.

Scheduling Considerations

Item	Information
Algebraic loops	There are algebraic loops that occur in SIL and PIL simulations but not in normal mode simulations: • **Single output/update function** in code generation optimization can introduce algebraic loops because the option introduces direct feedthrough via a combined output and update function. **Single output/update function** is not compatible with **Minimize algebraic loop occurrences** (in the Subsystem Parameters dialog box and **Configuration Parameters** > **Model Referencing** pane). **Minimize algebraic loop occurrences** allows code generation to remove algebraic loops by partitioning generated code between output and update functions to avoid direct feedthrough. • If you generate code for a virtual subsystem, code generation treats the subsystem as atomic and generates the code accordingly. The

Item	Information
	resulting code can change the execution behavior of your model, for example, by applying algebraic loops, and introduce inconsistencies to the simulation behavior. To enable consistent simulation and execution behavior for your model, declare virtual subsystems as atomic subsystems. For more information, see: • "Algebraic Loops" (Simulink) • "Algebraic Loops" (Simulink Coder) • "Code Generation of Subsystems" (Simulink Coder)
Exported functions in feedback loops	If your model has function-call subsystems and you export a subsystem that has context-dependent inputs (for example, feedback signals), the results of a SIL/PIL simulation with the generated code and the results of the normal mode simulation of your model can differ. One approach to make SIL/PIL and normal mode simulations yield identical results is to use Function-Call Feedback Latch (Simulink) blocks in your model. You can make context-dependent inputs become context-independent. Embedded Coder generates a warning identifying context-dependent inputs of exported function-call subsystems if you set **Configuration Parameters > Diagnostics > Connectivity > Context-dependent inputs** to one of the following: • `Enable all as warnings` • `Use local settings` • `Disable all` For more information, see: • "Code Generation of Subsystems" (Simulink Coder) • Function-Call Feedback Latch (Simulink) • "Context-dependent inputs" (Simulink)

Imported Data and Function Definitions

Item	Information
Imported data	In SIL and PIL simulations, you can use signals, parameters, and data stores that specify storage classes with imported data definitions. The simulations define storage for imported data associated with: • Signals at the root level of the component (on the I/O boundary) • Parameters • Global data stores SIL and PIL simulations do not define storage for other imported data. For example, the simulations do not define storage for imported data associated with: • Internal signals (not on the I/O boundary) • Local data stores In these cases, define the storage through custom code included by the component under test or through the PIL `rtw.pil.RtIOStreamApplicationFramework` API. See also "Tunable Parameters and SIL/PIL" on page 64-63.
`GetSet` custom storage class	SIL and PIL simulations support the `GetSet` custom storage class. The SIL/PIL test harness provides C definitions of the `Get` and `Set` functions that are used during simulations. For more information, see "Access Data Through Functions with Custom Storage Class `GetSet`" on page 23-92.
AUTOSAR Runtime Environment (RTE)	You can use top-model and Model block SIL/PIL and SIL/PIL block simulations to perform model-based testing of an AUTOSAR software component. The generated code for the AUTOSAR software component is linked with a basic component-specific AUTOSAR Runtime Environment (RTE) to create a test application. This application tests AUTOSAR API calls made by the AUTOSAR software component. **Note:** For Model block SIL/PIL, to test the AUTOSAR interface, set the **Code interface** block parameter to `Top model`.

Item	Information
	For more information, see “Verify AUTOSAR C Code with SIL and PIL”.

Related Examples

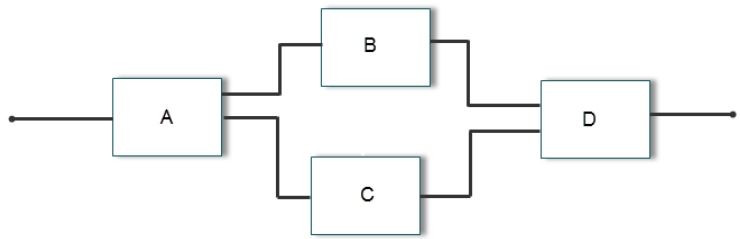
- “Test Generated Code with SIL and PIL Simulations”
- “Choose a SIL or PIL Approach” on page 64-11
- “Configure and Run SIL Simulation” on page 64-15
- “Check Configuration” on page 64-76

Choose a SIL or PIL Approach

In this section...

Consider a top model that consists of components `A`, `B`, `C`, and `D`:

- `A` and `B` are existing components for which code has previously been generated and tested.
- `C`, a referenced model, and `D`, a subsystem, are new components.



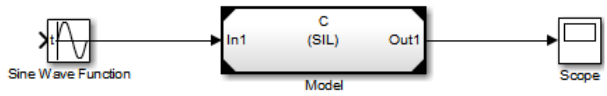
With software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulations, you can use the following approaches to numerical equivalence testing:

- Test code from all components together. See "Test Top-Model Code" on page 64-12.
- Test new components separately (before testing code from all components). See "Test Referenced Model Code" on page 64-13 and "Test Subsystem Code" on page 64-13.

For some forms of testing, you require a test harness model. The test harness model:

- Generates test vectors or stimulus inputs that feed the block under test.
- Makes it possible for you to observe or capture output from the block.

The following example shows a simple test harness model.



The block under test is a Model block. The Sine Wave block generates the input for the Model block. Through the Scope block, you can observe the output from the Model block.

Test Top-Model Code

To test code generated from the top-model components together (`A`, `B`, `C`, and `D`), you can use top-model SIL/PIL or Model block SIL/PIL.

- Top-model SIL/PIL:

  1. Create test vectors or stimulus inputs in the MATLAB workspace (Simulink).
  2. Run the top model in normal, SIL, and PIL simulation modes. The software loads the test vectors or stimulus inputs from the MATLAB workspace.
  3. For each simulation mode, observe or capture outputs.
  4. Verify numerical equivalence by comparing normal outputs against SIL and PIL outputs.

- Model block SIL/PIL:

  1. Create a Model block that contains the top-model components.
  2. Insert the Model block in a simulation model, for example, your test harness model.
  3. Run simulations, switching the Model block between normal, SIL, and PIL modes. For the SIL and PIL simulation modes, set the **Code interface** Model block parameter to `Top model`.
  4. Verify numerical equivalence by comparing normal outputs against SIL and PIL outputs.

Test Referenced Model Code

To test code generated from the component `C` as part of a model reference hierarchy, use the Model block SIL/PIL approach:

- Insert the Model block `C` in a simulation model, for example, your test harness model.
- Run simulations, switching the Model block between normal, SIL, and PIL modes. For the SIL and PIL simulation modes, set the **Code interface** Model block parameter to `Model reference`.
- Verify numerical equivalence by comparing normal outputs against SIL and PIL outputs.

Test Subsystem Code

To test code generated from the subsystem `D`, use the SIL or PIL block approach:

1. Insert the subsystem in a simulation model, for example, your test harness model.
2. Run a normal mode simulation, capturing the outputs.
3. Create a SIL or PIL block from the subsystem.
4. In the model, replace the subsystem with the SIL or PIL block.
5. Run a simulation of the model, capturing the outputs.
6. Verify numerical equivalence by comparing normal mode subsystem outputs against SIL or PIL block outputs.

Summary

Simulation Type	Component From Which Code Is Generated	Mode Selection Method	Generated Code Interface	Test Signal Source
Top-model SIL/PIL	Top model	Menu item on Simulink Editor toolbar	Standalone	MATLAB workspace (Simulink)
Model block SIL/PIL	Model referenced by Model block	Model block parameter **Simulation mode**	Determined by Model block parameter **Code interface**: standalone or model reference.	Simulation model, for example, test harness model

Simulation Type	Component From Which Code Is Generated	Mode Selection Method	Generated Code Interface	Test Signal Source
SIL or PIL block	Subsystem	Manual block substitution	Standalone	Simulation model, for example, test harness model.

Related Examples

- "Test Generated Code with SIL and PIL Simulations"
- "Configure and Run SIL Simulation" on page 64-15

More About

- "SIL and PIL Simulations" on page 64-2
- "Code Interfaces for SIL and PIL" on page 64-6

Configure and Run SIL Simulation

In this section...

There are three ways of running SIL and PIL simulations. You can use:

- The top model.
- Model blocks.
- SIL and PIL blocks that you create from subsystems.

Simulation with Top Model

To configure and run a top-model SIL or PIL simulation:

1. Open your model.
2. Select either **Simulation > Mode > Software-in-the-Loop (SIL)** or **Simulation > Mode > Processor-in-the-Loop (PIL)**. This option is available only if the model is configured for an ERT or AUTOSAR target. See "Model Configuration Parameters: Code Generation" (Simulink Coder) and "Export AUTOSAR Component XML and C Code" for configuration information.
3. If you have not already done so, in the Configuration Parameters dialog box, on the **Data Import/Export** pane:
   - In the **Input** check box and field, specify stimulus signals (or test vectors) for your top model.
   - Configure logging for model outputs, with either *output logging* or *signal logging*:

    - In the **Output** check box and field, specify *output logging*.
    - In the **Signal logging** check box and field, specify *signal logging*.
  - Disable logging of Data Store Memory variables. The software does not support this option for this simulation mode. If you do not clear the **Data stores** check box, the software produces a warning when you run the simulation.

4. If you are configuring a SIL simulation, specify the portable word sizes option. You can then switch seamlessly between the SIL and PIL modes. Select **Code Generation** > **Verification** > **Enable portable word sizes**.
5. If required, configure:
   - Code coverage.
   - Code execution profiling.
   - Creation of code generation report and static code metrics.
6. Start the simulation.

Note: On a Windows operating system, the Windows Firewall can potentially block your SIL simulation. To allow the SIL simulation, use the Windows Security Alert dialog box. For example, in Windows 7, click **Allow access**.

You cannot:

- Close the model while the simulation is running. To interrupt the simulation, in the Command Window, press **Ctrl+C**.
- Alter the model during the simulation. You can move blocks and lines as long as it does not alter the behavior of the model.

You can run a top-model SIL or PIL simulation with the command `sim(`*`model`*`)`. The software supports the `sim` command option `SrcWorkspace` for the value `'base'`.

For a PIL simulation, you control the way code compiles and executes in the target environment through connectivity configurations.

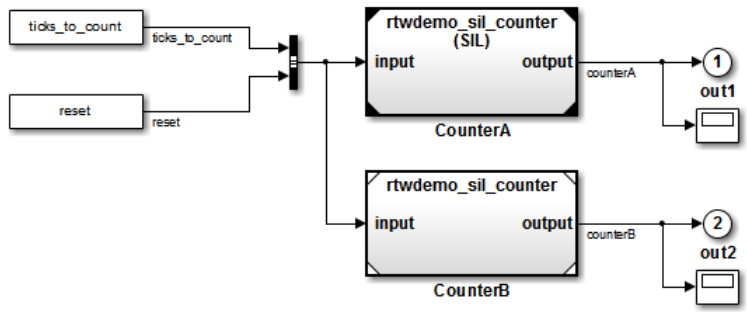
With a top-model SIL or PIL simulation, Simulink creates a hidden model, *`modelName`*`_wrapper`. The simulation generates code for the model and uses the hidden model to call this code at each time step. As a result, in some circumstances, logged signals can have a `_wrapper` suffix. The simulation can also generate warnings that refer to the hidden model. For example:

```
Warning: The model 'modelName_wrapper' has the 'Configuration Parameters' ...
```

Simulation with Model Blocks

To configure a Model block for a SIL or PIL simulation:

1. Open your model, for example, `rtwdemo_sil_modelblock`.
2. Right-click your Model block, for example, `Counter A`. In the context menu, select **Block Parameters (ModelReference)**, which opens the Function Block Parameters dialog box.
3. From the **Simulation Mode** drop-down list, select the required mode, for example, `Software-in-the-loop (SIL)`.
4. From the **Code interface** drop-down list, specify the code that you want to test, for example, `Model reference`.
5. Click **OK**. The software displays the simulation mode as a block label.



   If you select `Top model`, the software displays the block label `(SIL: Top)`.
6. If you are configuring a SIL simulation, specify the portable word sizes option. You can then switch seamlessly between the SIL and PIL modes. Select **Code Generation > Verification > Enable portable word sizes**.
7. If required, configure:
   - Code coverage.
   - Code execution profiling for your Model block, by configuring execution profiling for the top model.
   - Creation of code generation report and static code metrics.

8 Start the simulation.

Note: On a Windows operating system, the Windows Firewall can potentially block your SIL simulation. To allow the SIL simulation, use the Windows Security Alert dialog box. For example, in Windows 7, click **Allow access**.

For a PIL simulation, you control the way code compiles and executes in the target environment through connectivity configurations.

Simulation with Blocks From Subsystems

To create a SIL or PIL block from a subsystem and use this block to test the code generated from the subsystem:

1 In the Configuration Parameters dialog box, click the **All Parameters** tab.

2 From the **Create block** drop-down list, select either `SIL` or `PIL`.

3 If required, configure code execution profiling.

4 Click **OK**.

5 In your model window, right-click the subsystem that you want to simulate.

6 Select **C/C++ Code** > **Build This Subsystem**.

7 Click **Build**, which starts the subsystem build process that creates a SIL or PIL block for the generated subsystem code.

8 Add the generated block to an environment or test harness model that supplies test vectors or stimulus input.

9 Run simulations with the environment or test harness model.

Note: On a Windows operating system, the Windows Firewall can potentially block your SIL simulation. To allow the SIL simulation, use the Windows Security Alert dialog box. For example, in Windows 7, click **Allow access**.

You cannot create a SIL or PIL block if you do one of the following:

- Disable the `CreateSILPILBlock` property.
- Select a code coverage tool.

Create block appears dimmed.

For a PIL simulation, you control the way code compiles and executes in the target environment through connectivity configurations.

Configure Hardware Implementation Settings

For a SIL simulation, you must configure hardware implementation settings, which enables generated code compilation for your development computer. These settings can differ from the hardware implementation settings that you use when building the model for your production hardware. Use one of these approaches.

Approach	Details
Portable word sizes	Switch between SIL and PIL modes without regenerating code. You use the same generated source code files for the SIL simulation on your development computer and for production deployment on the target platform. To configure a model to use portable word sizes, in **Configuration Parameters** > **All Parameters**, set: • `ProdEqTarget` to `on`. • `PortableWordSizes` to `on`.

	When you generate code for a model with portable word sizes specified, the code generator conditionalizes data type definitions in `rtwtypes.h`:
	
	`#ifdef PORTABLE_WORDSIZES /* PORTABLE_WORDSIZES defined */`
	
	…
	
	`#else /* PORTABLE_WORDSIZES not defined */`
	
	…
	
	`#endif /* PORTABLE_WORDSIZES */`
	
	If you use the template makefile approach to build code for your development computer, the template makefile that you select controls the passing of the `PORTABLE_WORDSIZES` definition to the compiler. For example, `ert_unix.tmf` has the following lines:

Approach	Details

	`ifeq ($(PORTABLE_WORDSIZES),1)`
	`CPP_REQ_DEFINES += -DPORTABLE_WORDSIZES`
	`endif`
	
	Note: The template makefile that you use to build code for your target must not contain the `PORTABLE_WORDSIZES` definition.
	
	With the toolchain approach, the software specifies `-DPORTABLE_WORDSIZES` for the compiler only for host-based builds.
	
	For information about the template makefile and toolchain approaches to building code, see "Choose and Configure Build Process" (Simulink Coder).

	Consider the case where your target uses code that your development computer cannot compile. When you switch from the PIL mode to the SIL mode and try to simulate the model, you see compilation errors. You can try to work around this problem by adding the source code files to the `SkipForSil` group in the build information object `RTW.BuildInfo`. The SIL build on the host platform does not compile source files present in the `SkipForSil` group. For information about how you add source code files to a group in the build information object, see: • addSourceFiles (Simulink Coder) in the Simulink Coder documentation • "Customize Post-Code-Generation Build Processing" (Simulink Coder) in the Simulink Coder documentation

Approach	Details
	Numerical results can differ between generated code executing in a SIL simulation and generated code executing on the production hardware under one of these conditions: • Your model contains blocks implemented in TLC, for which C integral promotion in expressions can behave differently between the MATLAB host and the production hardware target. Normal and PIL simulation results match, but SIL simulation results can differ. • Your production hardware implements rounding to `Floor` for signed integer division, and divisions in your model use rounding mode `Ceiling`, `Floor`, `Simplest`, or `Zero`. Normal and PIL simulation results match, but SIL simulation results can differ. • You use custom code with the Stateflow product. In this case, type conversion statements are not inserted into the custom code, which target overflow behavior on the host can require. Normal and PIL simulation results match, but SIL simulation results can differ.
Test hardware	Use this approach only when you want to work around a limitation of portable word sizes. To configure a model for test hardware, in **Configuration Parameters** > **All Parameters**, set: • `PortableWordSizes` to `off`. • `ProdEqTarget` to `off`. • `TargetHWDeviceType` to `Custom Processor->MATLAB Host Processor`.
Production hardware	Use this approach only when the production hardware settings match your development computer architecture. In **Configuration Parameters** > **All Parameters**, set: • `PortableWordSizes` to `off`. • `ProdEqTarget` to `on`. • `ProdHWDeviceType` to match your development computer architecture. For example, you can select `Intel->x86-64 (Windows64)` and set `ProdLongLongMode` to `on`.

For information about test and production targets, see "Configure Run-Time Environment Options" (Simulink Coder) in the Simulink Coder documentation.

Log Internal Signals of a Component

SIL and PIL component outputs are available for observation and comparison with other simulation mode outputs. If you want to examine an internal signal, you can enable internal signal logging for top-model or Model block SIL or PIL. With signal logging, you can:

- Collect signal logging outputs during SIL/PIL simulations, for example, `logsout`.
- Log the internal signals and the root-level outputs of a SIL/PIL component.
- Manage the SIL/PIL signal logging settings with the Simulink Signal Logging Selector.
- Use the Simulation Data Inspector to:
  - Observe streamed signals during normal, SIL, and PIL simulations.
  - Compare logged signals from normal, SIL, and PIL simulations.

For SIL and PIL signal logging:

- Set **Configuration Parameters** > **All Parameters** > **Format** to `Dataset`.
- Select the **Configuration Parameters** > **Code Generation** > **Interface** > **Generate C API for: signals** check box.

  The C API determines the addresses of the internal signals that require logging.

You can use other methods to examine internal signals of the SIL or PIL component:

- Manually route the signal to the top level.
- Use global data stores to access internal signals:
  1. Inside the component, connect a Data Store Write block to the required signal.
  2. Outside the component, use a Data Store Read block to access the signal value.
- Use MAT-file logging. Note that:
  - MAT-file logging does not support signal logging. If signal logging is enabled, `logsout` is generated but not stored in the MAT-file.
  - For PIL, the target environment must support MAT-file logging.

For more information, see:

- "Test Points" (Simulink)
- "Export Signal Data Using Signal Logging" (Simulink)
- "Local and Global Data Stores" (Simulink)
- "Global Data Store Example" (Simulink)
- "Log Program Execution Results" (Simulink Coder)

Prevent Code Changes in Multiple Simulations

Use Model block SIL/PIL or the SIL/PIL block with fast restart when you want to run multiple SIL or PIL simulations with:

- Varying test vectors (parameter sets and input data).
- Unchanged generated code, that is, none of the simulations regenerate or rebuild code after the initial build. For example, you want to avoid the incremental code generation that an initial value change can trigger.

For Model block SIL/PIL, you can also use one of these methods:

- In your test harness model, set **Configuration Parameters** > **Model Referencing** > **Rebuild** to `Never`. If the Model block **Code interface** parameter is `Model reference`, the software does not rebuild the referenced model code. (If the **Code interface** parameter is `Top model`, the software ignores the **Rebuild** setting.)
- Create a protected model and generate source or binary code. Then, insert the protected model in your test harness model. With this method, you can verify top-model code (with the standalone code interface) or model reference code.

For the alternative methods of running Model block SIL/PIL, the following table summarizes code generation behavior after the initial build.

SIL and PIL Approach		Code Generation Behavior After Initial Build
Model block	**Configuration Parameters** > **Model Referencing** > **Rebuild** of test harness model set to `Never`.	**1** Component (algorithm) code from initial build is not regenerated. **2** Component code makefile is not called.

SIL and PIL Approach		Code Generation Behavior After Initial Build
		3 SIL/PIL application files from initial build are not regenerated. **4** SIL/PIL application makefile is called.
Model block (protected model)	Source code from protected model.	You observe the same behavior except for feature 2. In this case, the component code makefile is run. The component code is recompiled and linked to produce new object code.
	Binary code from protected model.	You observe features 1–4.

For more information, see:

- "Model Configuration Parameters: Model Referencing" (Simulink)
- "Create a Protected Model" (Simulink Coder)

Speed Up Testing

If your model has SIL/PIL blocks or Model blocks in SIL/PIL mode, you can speed up SIL/PIL testing by:

- Running the top-model simulation in accelerator mode (Simulink). This mode accelerates the simulation of model components that are not in SIL or PIL mode.
- Turning on fast restart (Simulink) with the **Fast restart** button on the Simulink Editor toolbar. After the first simulation, you can tune parameters and rerun simulations without model recompilation.

Note: The SIL and PIL simulation modes are not designed for the reduction of model simulation times. If you want to speed up the simulation of your model, use the rapid accelerator mode. For more information, see "What Is Acceleration?" (Simulink).

Simulation with Function Calls

Use the Simulink Function block and Function Caller block when you want to:

- Generate code that makes a function-call to external code, for example, driver or legacy code.
- Provide a subsystem that behaves like the external code in normal, SIL, or PIL simulations.

The example in "Configure Calls to AUTOSAR NVRAM Manager Service" shows how you can configure client calls to Basic Software (BSW) NVRAM Manager (NvM) service interfaces from your AUTOSAR software component. In a simulation, Simulink implements the BSW NvM calls through Simulink Function and preconfigured Function Caller blocks. For the final system, you link function-call stubs with external BSW function code that runs in the AUTOSAR Runtime Environment (RTE).

For more information, see:

- "Modeling Functions and Callers for Code Generation" on page 4-2
- "Generate Code for Functions and Callers" on page 4-6

Related Examples

- "SIL and PIL Simulations" on page 64-2
- "Choose a SIL or PIL Approach" on page 64-11
- "Test Generated Code with SIL and PIL Simulations"
- "Debug Generated Code During SIL Simulation" on page 64-37
- "View SIL and PIL Files in Code Generation Report" on page 64-59
- "Run Simulations Programmatically" (Simulink)
- "Simulation Mode Override Behavior in Model Reference Hierarchy" on page 64-35
- "SIL and PIL Limitations" on page 64-61
- "Configure Code Coverage with Third-Party Tools" on page 67-10
- "Code Execution Profiling with SIL and PIL" on page 58-2

Configure and Run PIL Simulation

In this section...

There are three ways of running SIL and PIL simulations. You can use:

- The top model.
- Model blocks.
- SIL and PIL blocks that you create from subsystems.

Simulation with Top Model

To configure and run a top-model SIL or PIL simulation:

1. Open your model.
2. Select either **Simulation > Mode > Software-in-the-Loop (SIL)** or **Simulation > Mode > Processor-in-the-Loop (PIL)**. This option is available only if the model is configured for an ERT or AUTOSAR target. See "Model Configuration Parameters: Code Generation" (Simulink Coder) and "Export AUTOSAR Component XML and C Code" for configuration information.
3. If you have not already done so, in the Configuration Parameters dialog box, on the **Data Import/Export** pane:
   - In the **Input** check box and field, specify stimulus signals (or test vectors) for your top model.
   - Configure logging for model outputs, with either *output logging* or *signal logging*:
     - In the **Output** check box and field, specify *output logging*.

    - In the **Signal logging** check box and field, specify *signal logging*.
  - Disable logging of Data Store Memory variables. The software does not support this option for this simulation mode. If you do not clear the **Data stores** check box, the software produces a warning when you run the simulation.

4. If you are configuring a SIL simulation, specify the portable word sizes option. You can then switch seamlessly between the SIL and PIL modes. Select **Code Generation > Verification > Enable portable word sizes**.
5. If required, configure:
   - Code coverage.
   - Code execution profiling.
   - Creation of code generation report and static code metrics.
6. Start the simulation.

Note: On a Windows operating system, the Windows Firewall can potentially block your SIL simulation. To allow the SIL simulation, use the Windows Security Alert dialog box. For example, in Windows 7, click **Allow access**.

You cannot:

- Close the model while the simulation is running. To interrupt the simulation, in the Command Window, press **Ctrl+C**.
- Alter the model during the simulation. You can move blocks and lines as long as it does not alter the behavior of the model.

You can run a top-model SIL or PIL simulation with the command `sim(`*`model`*`)`. The software supports the `sim` command option `SrcWorkspace` for the value `'base'`.

For a PIL simulation, you control the way code compiles and executes in the target environment through connectivity configurations.

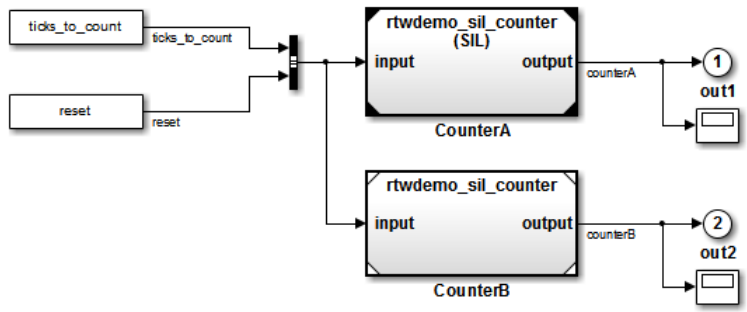
With a top-model SIL or PIL simulation, Simulink creates a hidden model, *`modelName`*`_wrapper`. The simulation generates code for the model and uses the hidden model to call this code at each time step. As a result, in some circumstances, logged signals can have a `_wrapper` suffix. The simulation can also generate warnings that refer to the hidden model. For example:

```
Warning: The model 'modelName_wrapper' has the 'Configuration Parameters' ...
```

Simulation with Model Blocks

To configure a Model block for a SIL or PIL simulation:

1. Open your model, for example, `rtwdemo_sil_modelblock`.
2. Right-click your Model block, for example, `Counter A`. In the context menu, select **Block Parameters (ModelReference)**, which opens the Function Block Parameters dialog box.
3. From the **Simulation Mode** drop-down list, select the required mode, for example, `Software-in-the-loop (SIL)`.
4. From the **Code interface** drop-down list, specify the code that you want to test, for example, `Model reference`.
5. Click **OK**. The software displays the simulation mode as a block label.



   If you select `Top model`, the software displays the block label `(SIL: Top)`.
6. If you are configuring a SIL simulation, specify the portable word sizes option. You can then switch seamlessly between the SIL and PIL modes. Select **Code Generation > Verification > Enable portable word sizes**.
7. If required, configure:
   - Code coverage.
   - Code execution profiling for your Model block, by configuring execution profiling for the top model.
   - Creation of code generation report and static code metrics.

8 Start the simulation.

Note: On a Windows operating system, the Windows Firewall can potentially block your SIL simulation. To allow the SIL simulation, use the Windows Security Alert dialog box. For example, in Windows 7, click **Allow access**.

For a PIL simulation, you control the way code compiles and executes in the target environment through connectivity configurations.

Simulation with Blocks From Subsystems

To create a SIL or PIL block from a subsystem and use this block to test the code generated from the subsystem:

1 In the Configuration Parameters dialog box, click the **All Parameters** tab.

2 From the **Create block** drop-down list, select either `SIL` or `PIL`.

3 If required, configure code execution profiling.

4 Click **OK**.

5 In your model window, right-click the subsystem that you want to simulate.

6 Select **C/C++ Code** > **Build This Subsystem**.

7 Click **Build**, which starts the subsystem build process that creates a SIL or PIL block for the generated subsystem code.

8 Add the generated block to an environment or test harness model that supplies test vectors or stimulus input.

9 Run simulations with the environment or test harness model.

Note: On a Windows operating system, the Windows Firewall can potentially block your SIL simulation. To allow the SIL simulation, use the Windows Security Alert dialog box. For example, in Windows 7, click **Allow access**.

You cannot create a SIL or PIL block if you do one of the following:

- Disable the `CreateSILPILBlock` property.
- Select a code coverage tool.

Create block appears dimmed.

For a PIL simulation, you control the way code compiles and executes in the target environment through connectivity configurations.

Log Internal Signals of a Component

SIL and PIL component outputs are available for observation and comparison with other simulation mode outputs. If you want to examine an internal signal, you can enable internal signal logging for top-model or Model block SIL or PIL. With signal logging, you can:

- Collect signal logging outputs during SIL/PIL simulations, for example, `logsout`.
- Log the internal signals and the root-level outputs of a SIL/PIL component.
- Manage the SIL/PIL signal logging settings with the Simulink Signal Logging Selector.
- Use the Simulation Data Inspector to:
  - Observe streamed signals during normal, SIL, and PIL simulations.
  - Compare logged signals from normal, SIL, and PIL simulations.

For SIL and PIL signal logging:

- Set **Configuration Parameters** > **All Parameters** > **Format** to `Dataset`.
- Select the **Configuration Parameters** > **Code Generation** > **Interface** > **Generate C API for: signals** check box.

  The C API determines the addresses of the internal signals that require logging.

You can use other methods to examine internal signals of the SIL or PIL component:

- Manually route the signal to the top level.
- Use global data stores to access internal signals:
  1. Inside the component, connect a Data Store Write block to the required signal.
  2. Outside the component, use a Data Store Read block to access the signal value.
- Use MAT-file logging. Note that:
  - MAT-file logging does not support signal logging. If signal logging is enabled, `logsout` is generated but not stored in the MAT-file.

- For PIL, the target environment must support MAT-file logging.

For more information, see:

- "Test Points" (Simulink)
- "Export Signal Data Using Signal Logging" (Simulink)
- "Local and Global Data Stores" (Simulink)
- "Global Data Store Example" (Simulink)
- "Log Program Execution Results" (Simulink Coder)

Prevent Code Changes in Multiple Simulations

Use Model block SIL/PIL or the SIL/PIL block with fast restart when you want to run multiple SIL or PIL simulations with:

- Varying test vectors (parameter sets and input data).
- Unchanged generated code, that is, none of the simulations regenerate or rebuild code after the initial build. For example, you want to avoid the incremental code generation that an initial value change can trigger.

For Model block SIL/PIL, you can also use one of these methods:

- In your test harness model, set **Configuration Parameters** > **Model Referencing** > **Rebuild** to `Never`. If the Model block **Code interface** parameter is `Model reference`, the software does not rebuild the referenced model code. (If the **Code interface** parameter is `Top model`, the software ignores the **Rebuild** setting.)
- Create a protected model and generate source or binary code. Then, insert the protected model in your test harness model. With this method, you can verify top-model code (with the standalone code interface) or model reference code.

For the alternative methods of running Model block SIL/PIL, the following table summarizes code generation behavior after the initial build.

SIL and PIL Approach		Code Generation Behavior After Initial Build
Model block	**Configuration Parameters** > **Model Referencing** > **Rebuild**	**1** Component (algorithm) code from initial build is not regenerated.

SIL and PIL Approach		Code Generation Behavior After Initial Build
	of test harness model set to `Never`.	**2** Component code makefile is not called. **3** SIL/PIL application files from initial build are not regenerated. **4** SIL/PIL application makefile is called.
Model block (protected model)	Source code from protected model.	You observe the same behavior except for feature 2. In this case, the component code makefile is run. The component code is recompiled and linked to produce new object code.
	Binary code from protected model.	You observe features 1–4.

For more information, see:

- "Model Configuration Parameters: Model Referencing" (Simulink)
- "Create a Protected Model" (Simulink Coder)

Speed Up Testing

If your model has SIL/PIL blocks or Model blocks in SIL/PIL mode, you can speed up SIL/PIL testing by:

- Running the top-model simulation in accelerator mode (Simulink). This mode accelerates the simulation of model components that are not in SIL or PIL mode.
- Turning on fast restart (Simulink) with the **Fast restart** button on the Simulink Editor toolbar. After the first simulation, you can tune parameters and rerun simulations without model recompilation.

Note: The SIL and PIL simulation modes are not designed for the reduction of model simulation times. If you want to speed up the simulation of your model, use the rapid accelerator mode. For more information, see "What Is Acceleration?" (Simulink).

Simulation with Function Calls

Use the Simulink Function block and Function Caller block when you want to:

- Generate code that makes a function-call to external code, for example, driver or legacy code.
- Provide a subsystem that behaves like the external code in normal, SIL, or PIL simulations.

The example in "Configure Calls to AUTOSAR NVRAM Manager Service" shows how you can configure client calls to Basic Software (BSW) NVRAM Manager (NvM) service interfaces from your AUTOSAR software component. In a simulation, Simulink implements the BSW NvM calls through Simulink Function and preconfigured Function Caller blocks. For the final system, you link function-call stubs with external BSW function code that runs in the AUTOSAR Runtime Environment (RTE).

For more information, see:

- "Modeling Functions and Callers for Code Generation" on page 4-2
- "Generate Code for Functions and Callers" on page 4-6

Related Examples

- "SIL and PIL Simulations" on page 64-2
- "Choose a SIL or PIL Approach" on page 64-11
- "Create PIL Target Connectivity Configuration" on page 64-40
- "Test Generated Code with SIL and PIL Simulations"
- "Configure Code Coverage with Third-Party Tools" on page 67-10
- "Code Execution Profiling with SIL and PIL" on page 58-2
- "View SIL and PIL Files in Code Generation Report" on page 64-59
- "Run Simulations Programmatically" (Simulink)
- "Simulation Mode Override Behavior in Model Reference Hierarchy" on page 64-35

- "SIL and PIL Limitations" on page 64-61

Simulation Mode Override Behavior in Model Reference Hierarchy

When the top model contains a Model block, the simulation mode of the top model can override the simulation mode of the Model block. The Model block itself can be a parent block containing child Model blocks at lower levels of its reference hierarchy. The simulation mode of the parent block can override the simulation mode of the child block.

You can specify the simulation mode of a top model to be normal, accelerator, rapid accelerator, SIL, or PIL. With a Model block, you can specify all modes *except* rapid accelerator. This table shows how the software determines the effective simulation mode of a Model block in a reference hierarchy.

Mode of Top Model or Parent Block	Mode of Parent or Child Block in Reference Hierarchy			
	Normal	Accelerator	SIL	PIL
Normal	Equivalent	Compatible	Compatible	Compatible
Accelerator	Override	Equivalent	Compatible if top model mode is accelerator. Error if parent block mode is accelerator.	Compatible if top model mode is accelerator. Error if parent block mode is accelerator.
Rapid accelerator	Override	Override	Error	Error
SIL	Override	Override	Equivalent	Error
PIL	Override	Override	Error	Equivalent

The different types of behavior are:

- Equivalent — Both parent and child Model block run in the same simulation mode.
- Compatible — The software simulates the child block in the mode specified for the child block, for example, when the simulation mode of the top model is normal or accelerator.
- Error — The simulation produces an error. For example, if a top model has simulation mode rapid accelerator but contains a child block in SIL or PIL mode, then running a simulation produces an error: the rapid accelerator mode cannot override the SIL and PIL mode of child blocks. This behavior avoids the risk of “false positives”, that is, the simulation of a model in rapid accelerator mode does not lead to the conclusion that generated source or object code of child Model blocks is tested or verified.

- Override — The simulation mode of the top model or parent Model block overrides the simulation mode of the child block. For example, if a top model or parent Model block that you configured for a SIL simulation contains a child Model block with normal or accelerator simulation mode, then the software simulates the child block in SIL mode. The override behavior:
  - Allows a Model block in the reference hierarchy to have the SIL or PIL mode.
  - Makes lower-level referenced models execute in SIL or PIL mode if you simulate the top model or parent Model block in SIL or PIL mode. You do not have to switch the simulation mode of every model component in the hierarchy.

For a model reference hierarchy that consists of multiple subhierarchies, if the top-model simulation mode is normal or accelerator, the software can run only one subhierarchy in PIL mode. For example, if your normal mode top model contains multiple Model blocks, you can specify the PIL mode for only one of the Model blocks.

Note: You can view your model hierarchy in the Model Dependency Viewer. In the Referenced Model Instances view, the software displays Model blocks differently to indicate their simulation modes, for example, normal, accelerator, SIL, and PIL. In this view, the software does not indicate the simulation mode of the top model.

More About

- "What Is Acceleration?" (Simulink)
- "SIL and PIL Simulations" on page 64-2

Debug Generated Code During SIL Simulation

If a software-in-the-loop (SIL) simulation fails or you notice differences between the outputs of your original functions and the generated code, you can rerun the SIL simulation with a debugger enabled. By inserting breakpoints, you can observe the behavior of code sections, which can help you to understand the cause of the issue.

For a SIL simulation failure, you can also view information from the standard output and standard error streams in the Diagnostic Viewer. For example:

- Output from `printf` statements in your code.
- Error messages sent to `stderr`.
- Some low-level system messages.

During a SIL simulation, the SIL application redirects the `stdout` and `stderr` streams. When the application terminates, the Diagnostic Viewer displays the information from the redirected streams. The SIL application also provides a basic signal handler, which captures the POSIX® signals `SIGFPE`, `SIGILL`, `SIGABRT`, and `SIGSEV`. For this signal handler, the SIL application includes the file `signal.h`.

A SIL simulation supports these debuggers;

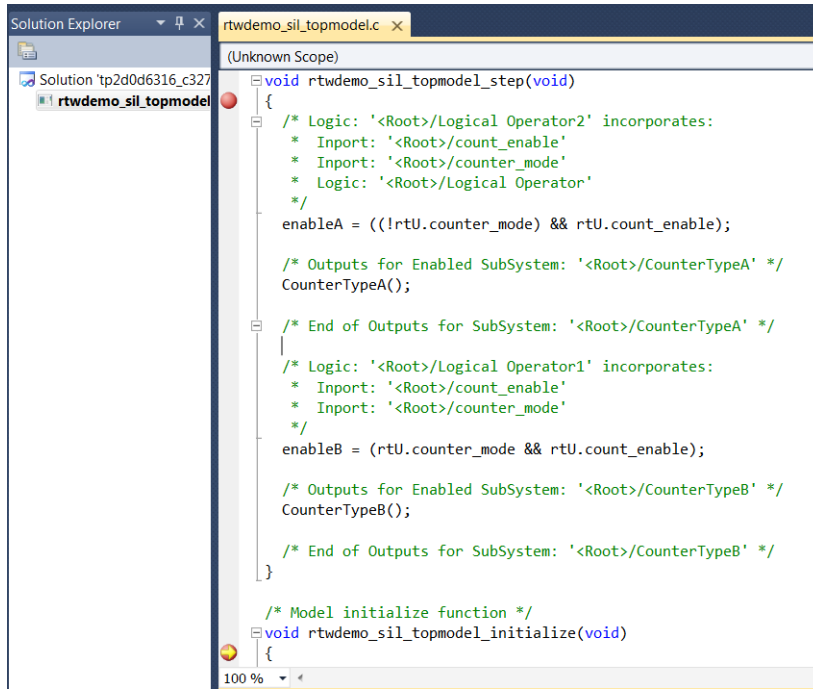
- On Windows, Microsoft Visual Studio® debugger.
- On Linux, GNU Data Display Debugger (DDD).

Note: You can perform SIL debugging only if the Simulink product family supports your Microsoft Visual C++ or GNU GCC compiler. For more information, see supported compilers.

To enable your debugger for a SIL simulation, on the **Configuration Parameters** > **Code Generation** > **Verification** pane, select the **Enable source-level debugging for SIL** check box.

If your top model has Model blocks where the **Code interface** block parameter is set to `Top model`, then the **Enable source-level debugging for SIL** parameters for the top model and referenced models must have the same settings. Otherwise, the software produces an error.

When you run the SIL simulation, for example on a Windows computer, your *`model`*`.c` or *`model`*`.cpp` file opens in the Microsoft Visual Studio IDE with debugger breakpoints at the start of the *`model`*`_initialize` and *`model`*`_step` functions.



You can now use the debugger features to observe code behavior. For example, you can step through code and examine variables.

To end the debugging session:

1. Remove all breakpoints.
2. Click the `Continue` button (**F5**).

   The SIL simulation runs to completion and the Microsoft Visual Studio IDE closes.

Note: In the Microsoft Visual Studio IDE, if you select **Debug > Stop Debugging**, the SIL simulation times out with the following error message:

```
The timeout of 1 seconds for receiving data from the rtiostream
```

```
interface has been exceeded. There are multiple possible causes
for this failure.
...
...
```

Related Examples

- “Configure and Run SIL Simulation” on page 64-15

Create PIL Target Connectivity Configuration

In this section...

Target Connectivity Configurations for PIL

Use target connectivity configurations and the target connectivity API to customize processor-in-the-loop (PIL) simulation for your target environments.

Through a target connectivity configuration, you specify:

- A configuration name for a target connectivity API implementation.
- Settings that define the set of compatible Simulink models. For example, the set of models that have a particular system target file, template makefile, and hardware implementation.

A PIL simulation requires a target connectivity API implementation that integrates third-party tools for:

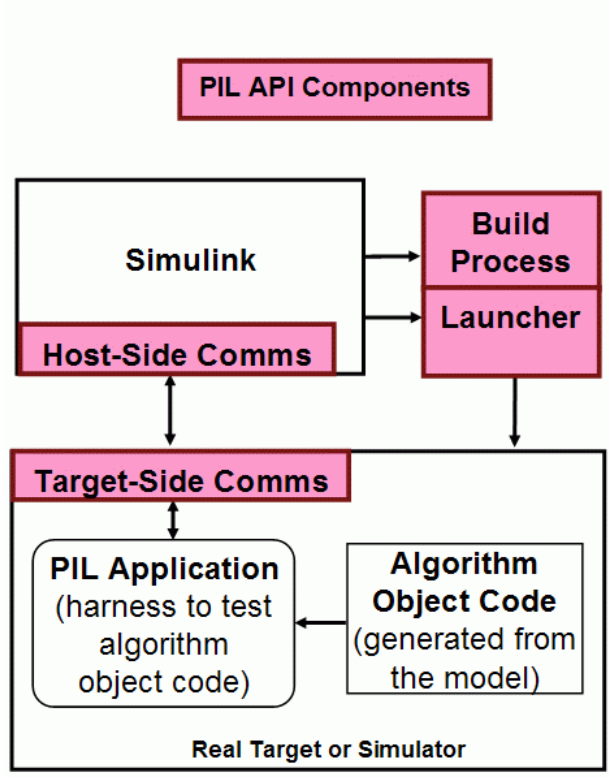
- Cross-compiling generated code, creating the PIL application that runs on the target hardware.
- Downloading, starting, and stopping the application on the target.
- Communicating between Simulink and the target.

You can have many different target connectivity configurations for PIL simulation. Register a connectivity configuration with Simulink by creating an `sl_customization.m` file and placing it on the MATLAB search path.

When you run a PIL simulation, the software determines which of the available connectivity configurations to use. The software looks for a connectivity configuration that is compatible with the model under test. If the software finds multiple or no compatible connectivity configurations, the software generates an error message with information about resolving the problem.

Create a Target Connectivity API Implementation

This diagram shows the components of the PIL target connectivity API.



You must provide implementations of the three API components:

- Build API — Specify the Simulink Coder toolchain or template makefile approach for building generated code.
- Launcher API — Control how Simulink starts and stops the PIL executable.
- Communications API — Customize connectivity between Simulink and the PIL target. Embedded Coder provides host-side support for TCP/IP and serial communications, which you can adapt for other protocols.

These steps outline how you create a target connectivity API implementation. The example code shown in the steps is taken from `ConnectivityConfig.m` in "Configure Processor-In-The-Loop (PIL) for a Custom Target".

1. Create a subclass of `rtw.connectivity.Config`.

   ```
   ConnectivityConfig < rtw.connectivity.Config
   ```

2. In the subclass:

   - Instantiate `rtw.connectivity.MakefileBuilder`, which configures the build process.

     ```
     builder = rtw.connectivity.MakefileBuilder(componentArgs, ...
                                                targetApplicationFramework, ...
                                                exeExtension);
     ```

   - Create a subclass of `rtw.connectivity.Launcher`, which downloads and executes the application using a third-party tool.

     ```
     launcher = mypil.Launcher(componentArgs, builder);
     ```

3. Configure your `rtiostream` API implementation of the host-target communications on page 64-46 channel.

   - For the target side, you must provide the driver code for communications, for example, TCP/IP or serial communications. To integrate this code into the build process, create a subclass of `rtw.pil.RtIOStreamApplicationFramework`.
   - For the host side, you can use a supplied library for TCP/IP or serial communications. Instantiate `rtw.connectivity.RtIOStreamHostCommunicator`, which loads and initializes the library that you specify.

     ```
     hostCommunicator = rtw.connectivity.RtIOStreamHostCommunicator(componentArgs, ...
                                                                    launcher, ...
                                                                    rtiostreamLib);
     ```

4. If you require execution-time profiling of generated code, create a timer object that provides details of the hardware-specific timer and associated source files. See "Specify Hardware Timer" on page 64-52.

Note: Each time you modify a connectivity implementation, close and reopen the models to refresh them.

Register a Connectivity API Implementation

To register a target connectivity API implementation as a target connectivity configuration in Simulink:

1. Create or update an `sl_customization.m` file. In this file:
   - Create a target connectivity configuration object that specifies, for example, the configuration name for a target connectivity API implementation and compatible models.
   - Invoke `registerTargetInfo`.
2. Add the folder containing `sl_customization.m` to the search path and refresh your customizations.

   ```
   addpath(sl_customization_path);
   sl_refresh_customizations;
   ```

For more information, see `rtw.connectivity.ConfigRegistry`.

Verify Target Connectivity Configuration

To verify your target connectivity configuration early on and independently of your model development and code generation, use the supplied `piltest` function. With the function, you can run a suite of tests. In the tests, the function runs various normal, SIL, and PIL simulations. The function compares results and produces errors if it detects differences between simulation modes.

Target Connectivity API Examples

For step-by-step examples, see:

- "Configure Processor-In-The-Loop (PIL) for a Custom Target"

  This example shows you how to create a custom PIL implementation using the target connectivity APIs. You can examine the code that configures the build process to support PIL, a downloading and execution tool, and a communication channel between host and target. To activate a full host-based PIL configuration, follow the steps in the example.
- "Create a Target Communication Channel for Processor-In-The-Loop (PIL) Simulation"

This example shows you how to implement a communication channel for use with the Embedded Coder product and your embedded target. This communication channel enables exchange of data between different processes. PIL simulation requires exchange of data between the Simulink software running on your development computer and deployed code executing on target hardware.

The `rtiostream` interface provides a generic communication channel that you can implement in the form of target connectivity drivers for a range of connection types. The example shows how to configure your own target-side driver for TCP/IP, to operate with the default host-side TCP/IP driver. The default TCP/IP communications allow high-bandwidth communication between host and target, which you can use for transferring data such as video.

Note: If you customize the `rtiostream` TCP/IP implementation for your PIL simulations, you must turn off Nagle's algorithm for the server side of the connection. If Nagle's algorithm is not turned off, your PIL simulations can run at a significantly slower speed. The *`matlabroot`*`/rtw/c/src/rtiostream/rtiostreamtcpip/rtiostream_tcpip.c` file shows how you can turn off Nagle's algorithm:

```
/* Disable Nagle's Algorithm*/
option = 1;
sockStatus = setsockopt(lFd,IPPROTO_TCP,TCP_NODELAY,(char*)&option,sizeof(option));
```

The code for your custom TCP/IP implementation can require modification.

The example also shows how to implement custom target connectivity drivers, for example, using serial, CAN, or USB for both host and target sides of the communication channel.

See Also

`piltest` | `rtw.connectivity.Config` | `rtw.connectivity.ConfigRegistry` | `rtw.connectivity.Launcher` | `rtw.connectivity.MakefileBuilder` | `rtw.connectivity.RtIOStreamHostCommunicator` | `rtw.pil.RtIOStreamApplicationFramework`

Related Examples

- “Specify Hardware Timer” on page 64-52
- “Subclass Constructors” (MATLAB)

More About

- "Host-Target Communication for PIL" on page 64-46

Host-Target Communication for PIL

In this section...

Communications `rtiostream` API

The `rtiostream` API supports communications for the target connectivity API. Use the `rtiostream` API to implement a communication channel that enables data exchange between different processes.

PIL simulation requires a host-target communications channel. This communications channel comprises driver code that runs on the host and target. The `rtiostream` API defines the signature of target-side and host-side functions that this driver code must implement.

The API is independent of the physical layer that sends the data. Possible physical layers include RS232, Ethernet, or Controller Area Network (CAN).

A full `rtiostream` implementation requires both host-side and target-side drivers. Code generation software includes host-side drivers for the default TCP/IP implementation and a version for serial communications. To use:

- The TCP/IP `rtiostream` communications channel, you must provide, or obtain from a third party, target-specific TCP/IP device drivers.
- The serial communications channel, you must provide, or obtain from a third party, target-specific serial device drivers.

For other communication channels and platforms, the code generation software does not provide default implementations. You must provide both the host-side and the target-side drivers.

The `rtiostream` API comprises the following functions:

- `rtIOStreamOpen`

- `rtIOStreamSend`
- `rtIOStreamRecv`
- `rtIOStreamClose`

For information about:

- Using `rtiostream` functions in a connectivity implementation, see "Create a Target Connectivity API Implementation" on page 64-41.
- Testing the `rtiostream` shared library methods from MATLAB code, see `rtiostream_wrapper`.
- Debugging and verifying the behavior of custom `rtiostream` interface implementations, see "Test an `rtiostream` Driver" on page 64-48.

Synchronize Host and Target

If you use the `rtiostream` API to implement the communications channel, the host and target must be synchronized, which prevents Simulink from transmitting and receiving data before the target application is fully initialized.

To synchronize the host and target for TCP/IP rtiostream implementations, use the `setInitCommsTimeout` method from `rtw.connectivity.RtIOStreamHostCommunicator`. This approach works well for connection-oriented TCP/IP `rtiostream` implementations because Simulink automatically waits until the target server is running.

With other `rtiostream` implementations, for example, serial, the Simulink side of the `rtiostream` connection opens without waiting for the target to be fully initialized. In this case, you must make your `Launcher` implementation wait until the target application is fully initialized. Use one of the following approaches to synchronize your host and target:

- Add a pause at the end of the `Launcher` implementation that makes the `Launcher` wait until target initialization is complete.
- In the `Launcher` implementation, use third-party downloader or debugger APIs that wait until target initialization is complete.
- Implement a handshaking mechanism in the `Launcher` / `rtiostream` implementation that confirms completion of target initialization.

Test an `rtiostream` Driver

Use a test suite to debug and verify the behavior of custom `rtiostream` interface implementations.

The test suite has the following advantages:

- Reduces time for integrating custom hardware that does not have built-in `rtiostream` support.
- Reduces time for testing custom `rtiostream` drivers.
- Helps analyze the performance of custom `rtiostream` drivers.

The test suite has two parts. One part of the test suite runs on the target.

Note: After building the target application, download it to the target and run it.

To start this part, compile and link the following files, which are in the folder *`matlabroot`*`/toolbox/coder/rtiostream/src/rtiostreamtest` (open).

- `rtiostreamtest.c`
- `rtiostreamtest.h`
- `rtiostream.h`, located in the folder *`matlabroot`*`/rtw/c/src` (open)
- `rtiostream` implementation under investigation (for example, `rtiostream_tcpip.c`)
- `main.c`

To run the MATLAB part of the test suite, invoke `rtiostreamtest`. The syntax is as follows:

```
rtiostreamtest(connection,param1,param2)
```

- `connection` is a character vector indicating the communication method. It can have values `'tcp'` or `'serial'`.
- `param1` and `param2` have different values depending on the value of `connection`.
  - If `connection` is `'tcp'`, then `param1` and `param2` are hostname and port, respectively. For example, `rtiostreamtest('tcp', 'localhost', 2345)`.

- If `connection` is `'serial'`, then `param1` and `param2` are COM port and baud rate, respectively. For example, `rtiostreamtest('serial', 'COM1', 9600)`.

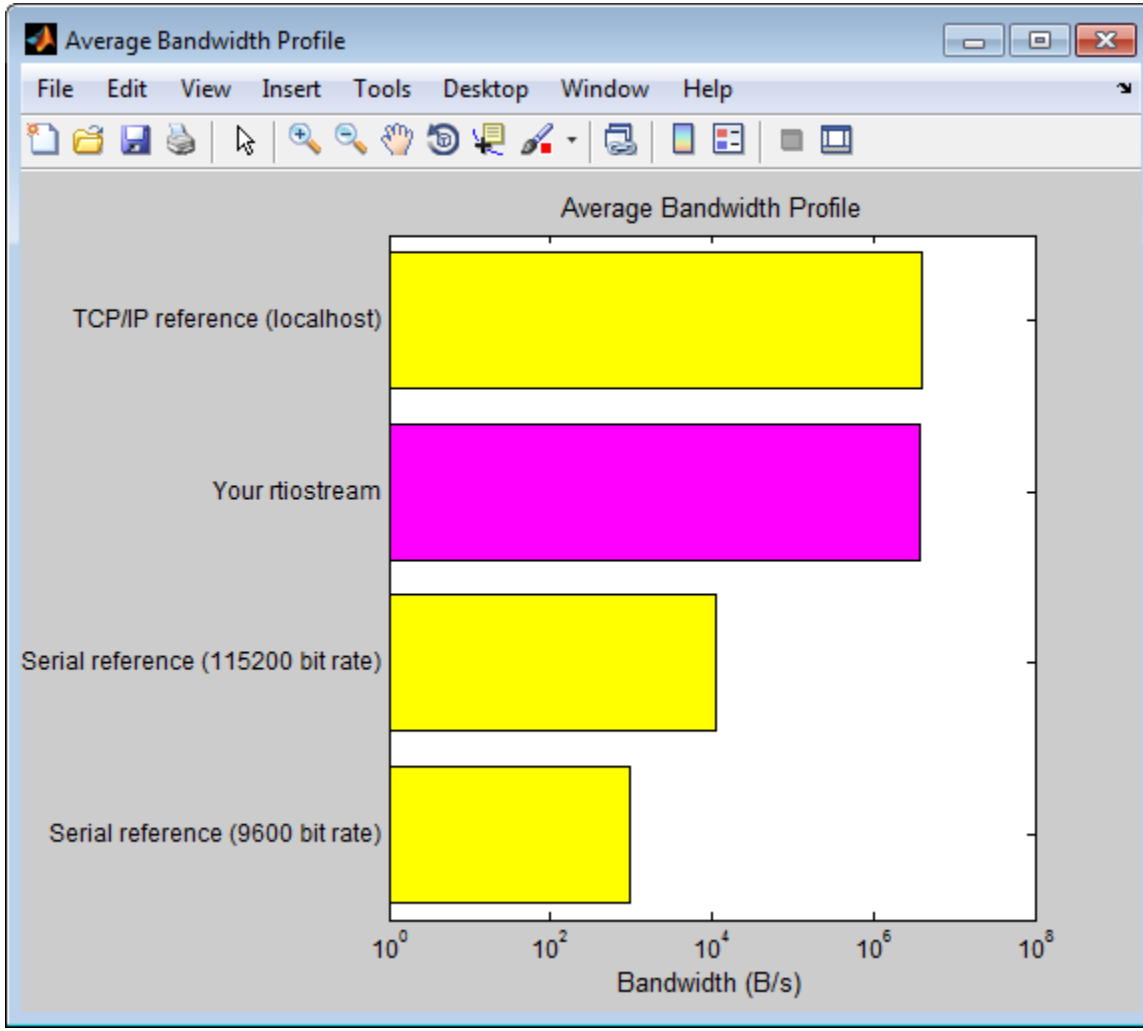
You can run the MATLAB part of the test suite as follows:

```
rtiostreamtest('tcp','localhost','2345')
```

An output in the following format appears in the MATLAB window:

```
### Test suite for rtiostream ###
Initializing connection with target...

### Hardware characteristics discovered
Size of char    : 8 bit
Size of short   : 16 bit
Size of int     : 32 bit
Size of long    : 32 bit
Size of float   : 32 bit
Size of double  : 64 bit
Size of pointer : 64 bit
Byte ordering   : Little Endian

### rtiostream characteristics discovered
Round trip time : 0.96689 ms
rtIOStreamRecv behavior : non-blocking

### Test results
Test 1 (fixed size data exchange): ......... PASS
Test 2 (varying size data exchange): ......... PASS

### Test suite for rtiostream finished successfully ###
```

Furthermore, the following profile appears.



See Also

`rtiostream_wrapper` | `rtIOStreamClose` | `rtIOStreamOpen` | `rtIOStreamRecv` | `rtIOStreamSend` | `rtw.connectivity.RtIOStreamHostCommunicator`

Related Examples

- “Create PIL Target Connectivity Configuration” on page 64-40

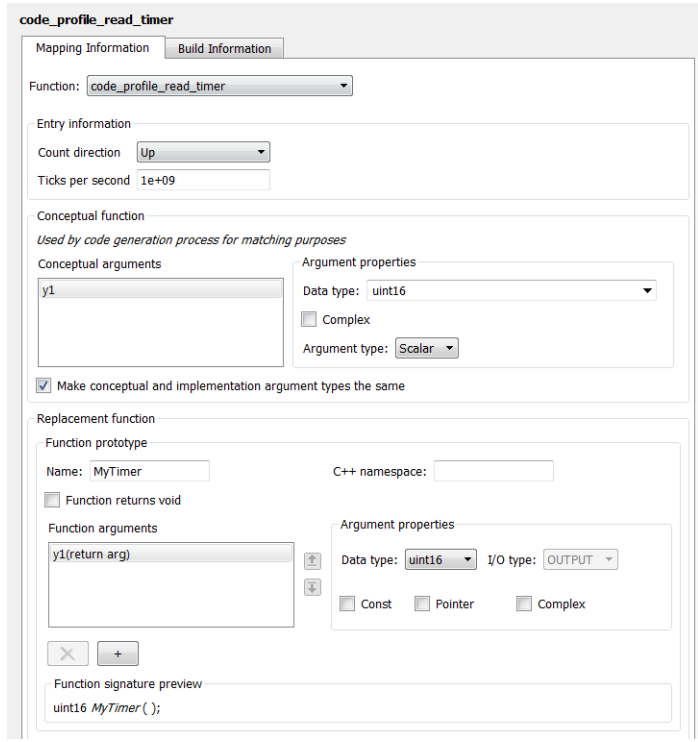
Specify Hardware Timer

For processor-in-the-loop (PIL) code execution profiling, you must create a timer object that provides details of the hardware-specific timer and associated source files. You can use the Code Replacement Tool or the code replacement library API to specify this hardware-specific timer.

To specify the timer with the Code Replacement Tool:

1. Open the Code Replacement Tool. In the Command Window, enter `crtool`.
2. Create a new code replacement table. Select **File** > **New table**.
3. Create a new function entry. Under **Tables List**, right-click the new table. Then, from the context-menu, select **New entry** > **Function**.
4. In the middle view, select the new unnamed function.
5. On the **Mapping Information** pane:
   a. From the **Function** drop-down list, select `code_profile_read_timer`.
   b. Specify the count direction for your timer. For example, from the **Count direction** drop-down list, select `Up`.
   c. In the **Ticks per second** field, specify the number of ticks per second for your timer, for example, `1e+09`.

      The default value is 0. In this case, the software reports time measurements in terms of ticks, not seconds.
   d. In the **Name** field, specify a replacement function name, for example, `MyTimer`.
   e. Click **Apply**.



f To validate the function entry, click **Validate entry**.

6 On the **Build Information** pane, specify the required build information. See "Specify Build Information for Replacement Code" on page 51-59.

7 Save the table (**Ctrl+S**). When you save the table for the first time, use the Save As dialog box to specify the file name and location.

You must save the table in a location that is on the MATLAB search path. For example, you can save this file in the folder for your subclass of `rtw.connectivity.Config`.

The software stores your timer information as a code replacement library table.

8 Assuming you save the table as *`MyCrlTable.m`*, in your subclass of `rtw.connectivity.Config`, add the following line:

```
setTimer(this, MyCrlTable)
```

Related Examples

- “Create a Target Connectivity API Implementation” on page 64-41
- “Code Execution Profiling with SIL and PIL” on page 58-2
- “Specify Build Information for Replacement Code” on page 51-59

More About

- “What Is Code Replacement?” on page 38-2
- “What Is Code Replacement Customization?” on page 51-3

PIL Simulation Sequence

A processor-in-the-loop (PIL) simulation cross-compiles production source code, and then downloads and runs object code on your target hardware. The connectivity configuration that you create controls the way code is compiled and executed on the target. This table describes the sequence of stages in a PIL simulation.

Stage		Description
1	Start	For top-model PIL, on the Simulink Editor toolbar, you select the `Processor-in-the-Loop (PIL)` mode, and then click the Run button. For Model block PIL, you set the **Simulation mode** parameter of the Model block to `Processor-in-the-loop (PIL)`, and then run a simulation of the harness model that contains the Model block. For the PIL block, you run a simulation of the harness model that contains the PIL block.
2	Validate target connectivity	The software verifies that a target connectivity configuration is registered for PIL. Otherwise, the software produces an error.
3	Generate production source code and build object code for target	The generated source code is identical to the code that is produced when you run the `slbuild` command. • For top-model PIL or Model block PIL with block parameter **Code interface** set to `Top model`, the generated code is identical to the code produced when you run `slbuild('model')`. • For Model block PIL with block parameter **Code interface** set to `Model reference`, the generated code is identical to the code produced when you run `slbuild('model', 'ModelReferenceRTWTarget')`. The model reference simulation target is also produced. The software builds object code for the target by using the template makefile or toolchain that you specify.
4	Create instances of PIL API components	The software instantiates your `rtw.connectivity.Config` class, which creates

Stage		Description
		instances of `rtw.connectivity.MakefileBuilder`, `rtw.connectivity.Launcher`, `rtw.pil.RtIOStreamApplicationFramework`, and `rtw.connectivity.RtIOStreamHostCommunicator`.
5	Generate PIL files	The generated PIL files are in the `pil` folder. At the end of the simulation, use the code generation report to view the files.
6	Build target application	The software: • Uses your instance of `rtw.connectivity.MakefileBuilder` to build the target application. • Compiles the PIL interface file, `xil_interface.c`, and other PIL files into the target executable file. On a Windows system, for example, this file is called *`modelName`*`.exe`. The object code, including the executable file, is in the `pil` folder. • If configured, produces the code generation report.
7	Start target application	The software uses `rtw.connectivity.Launcher` to start the application on the target.
8	Simulink engine interacts with PIL S-function	The Simulink engine interacts with the PIL S-function in the same way that it interacts with a C S-function. From the host-side, the PIL S-function communicates with the target executable code through `rtIOStream` commands. On the target side, `xil_interface` executes generated code.
9	Stop target application	The software uses `rtw.connectivity.Launcher` to stop the application on the target.

Stage		Description
10	End PIL simulation	For top-model PIL, at the end of the simulation, the software destroys the `rtw.connectivity.Config` instance. For Model block PIL and PIL block, the block creates and owns the `rtw.connectivity.Config` instance, which is not destroyed at the end of the simulation. You can rerun the simulation, which now does not require the creation of another `rtw.connectivity.Config` instance. If you want to destroy the instance, close the parent model.

See Also

`rtw.connectivity.Config` | `rtw.connectivity.Launcher` | `rtw.connectivity.MakefileBuilder` | `rtw.connectivity.RtIOStreamHostCommunicator` | `rtw.pil.RtIOStreamApplicationFramework`

Related Examples

- "Create PIL Target Connectivity Configuration" on page 64-40
- "View SIL and PIL Files in Code Generation Report" on page 64-59

More About

- "SIL and PIL Simulations" on page 64-2
- "Simulink Engine Interaction with C S-Functions" (Simulink)

Verification of Code Generation Assumptions

The settings on the **Configuration Parameters > Hardware Implementation** pane specify target behavior, which result in the implementation of implicit assumptions in the generated code. Incorrect settings can lead to:

- Suboptimal code
- Code execution failure, incorrect code output, and nondeterministic code behavior

At the start of a processor-in-the-loop (PIL) simulation, the software verifies the **Hardware Implementation** pane settings with reference to the target hardware. The software checks:

- The correctness of settings. For example, the integer bit length in the **Number of bits: int** field.
- Whether the settings are optimized. For example, the rounding of signed integer division in the **Signed integer division rounds to** field.

If required, the software generates warnings and errors.

See Also

"Hardware Implementation Pane" (Simulink)

More About

- "SIL and PIL Simulations" on page 64-2

View SIL and PIL Files in Code Generation Report

With top-model and Model block SIL and PIL simulations, you can produce a code generation report and static code metrics that cover SIL and PIL files. The information helps you to:

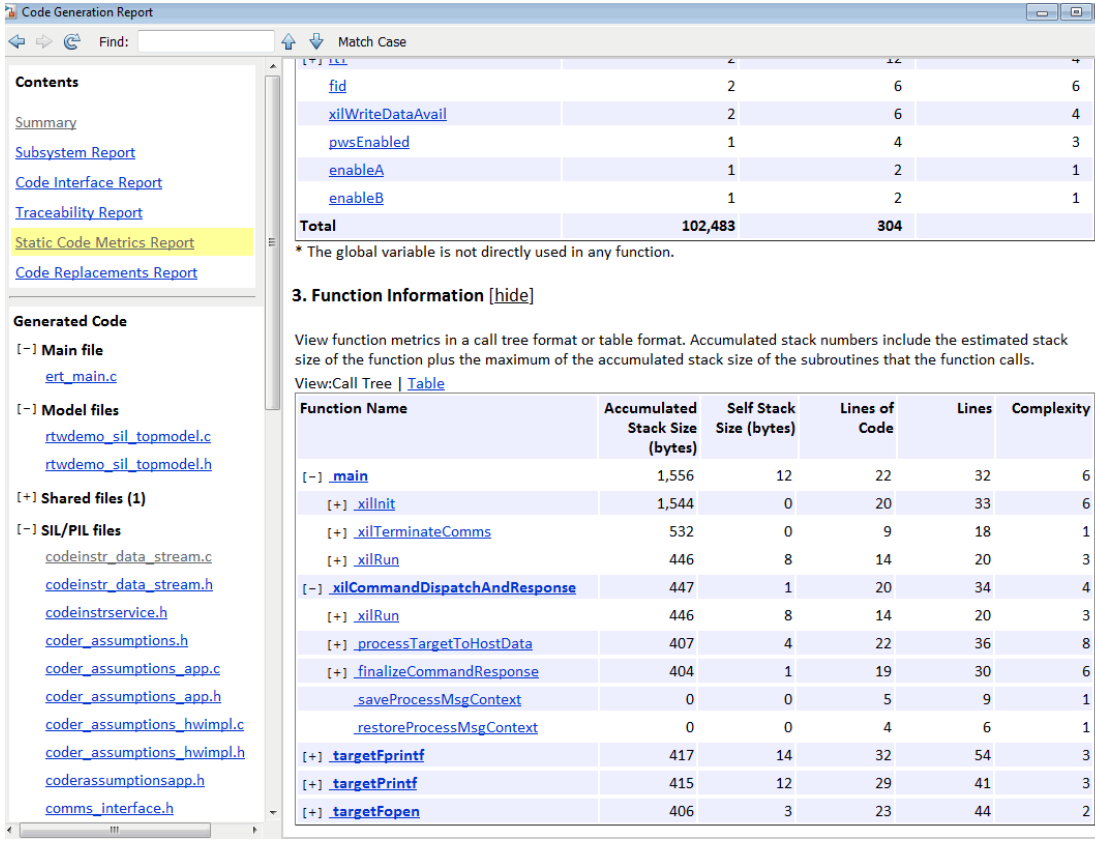
- Understand and review the SIL and PIL testing process.
- See how your registered custom target connectivity files fit into the target application that runs during a SIL or PIL simulation.

This capability is not supported for simulations that you run with the PIL block.

To configure the creation of a code generation report and static code metrics, on the **Configuration Parameters > Code Generation > Report** pane, select the **Create code generation report**, **Open report automatically**, and **Static code metrics** check boxes. Then click **OK**.

At the end of the simulation, in the Code Generation Report window:

- To review code metrics, in the **Contents** view, click **Static Code Metrics Report**.
- To review SIL and PIL files, in the **Generated Code** view, expand the **SIL/PIL files** node. For example:
  - To review the S-function that runs on the host, click *`modelName`*`_sbs.c` or *`modelName`*`_pbs.c`.
  - To view the SIL or PIL interface that runs on the target, click `xil_interface.c`.



Note: Do not use the SIL or PIL files in code development as these files can change over releases. Use supplied APIs for code development.

Related Examples

- “Static Code Metrics” on page 35-34

More About

- “HTML Code Generation Report Extensions” on page 35-3

SIL and PIL Limitations

In this section...

About SIL and PIL Limitations

With Embedded Coder, you can run software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulations in three ways:

- Top-model SIL/PIL — Set the top-model simulation mode to `Software-in-the-Loop (SIL)` or `Processor-in-the-Loop (PIL)`.
- Model block SIL/PIL — Set the Model block parameter **Simulation mode** to `Software-in-the-loop (SIL)` or `Processor-in-the-loop (PIL)`.
- SIL/PIL block — Use SIL or PIL blocks in the model.

The following sections describe modeling and code generation features that are either unsupported or partially supported by SIL and PIL simulations.

General SIL and PIL Limitations

Tunable Parameters and SIL/PIL

For Model block SIL/PIL and SIL/PIL block simulations, you can tune tunable *workspace* parameters but not tunable *dialog box* parameters. For information about tuning parameters, see "Block Parameter Representation in the Generated Code" on page 19-47.

For a top model with tunable parameters, you can run a SIL/PIL simulation but you cannot tune the parameters during the simulation.

The software cannot define, initialize, or tune the following types of tunable workspace parameters.

Parameter description	Software response		
	Top-Model SIL/PIL	Model Block SIL/PIL	SIL/PIL Block
Parameters with storage class that applies `"static"` scope or `"const"` keyword. For example, `Custom`, `Const`, or `ConstVolatile`	Warning	Warning	Warning
Parameters with multiword, fixed-point data types	Warning	Error	Warning
Parameters with data types that have different sizes on host and target	Warning	Error	Warning

For C++ class code, SIL/PIL you can tune tunable workspace parameters when **Parameter visibility** is `public`. If **Parameter visibility** is `private` or `protected`, tuning is supported only if **Parameter access** is `Method` or `Inlined method`.

For top-model SIL/PIL and the SIL/PIL block, consider the case where all of the following conditions apply:

- **Code Generation > Interface > Code interface packaging** is `Reusable function`.
- **All Parameters > Use dynamic memory allocation for model initialization** is not selected.

- **Optimization > Signals and Parameters > Default parameter behavior** is `Tunable`.
- The model contains parameters with storage class `Auto` or `SimulinkGlobal`.

If the SIL/PIL component cannot dynamically initialize tunable parameters in the `rtP` model parameter structure, you see an error message like the following:

```
Parameter Dialog:InitialOutput in 'rtwdemo_sil_topmodel/CounterTypeA/count'
is part of the imported "rtP" structure in the generated code but cannot be
initialized by SIL or PIL. To avoid this error, make sure the parameter
corresponds to a tunable base workspace variable with a storage class such
as SimulinkGlobal and is supported for dynamic parameter initialization /
tuning with SIL/PIL. Alternatively, select Configuration Parameters >
Code Generation > Interface and set 'Code interface packaging' to
'Nonreusable function', or select 'Use dynamic memory allocation for model
initialization'.
```

If you select **All Parameters > Use dynamic memory allocation for model initialization**, this limitation does not apply.

For Model block SIL/PIL, if you specify the code under test to be `Top model`, you can tune parameters while a simulation runs. If you tune parameters between successive runs of the simulation, the software generates new code for the later run. The new code uses your latest settings as initial parameter values.

Global and Local Data Stores

SIL/PIL supports global data stores. For components that are not export-function models, top-model SIL/PIL and SIL/PIL block simulations that access global data stores must be single rate. Otherwise, the software produces an error.

SIL/PIL does not support local data stores.

SIL/PIL Does Not Check Simulink Coder Error Status

SIL/PIL does not check the Simulink Coder error status of the generated code under test. This error status flags exceptional conditions during execution of the generated code.

Blocks in the model can also set the Simulink Coder error status, for example, custom blocks that you create. SIL/PIL does not check this error status and report errors.

Missing Code Interface Description File Errors

SIL/PIL requires a code interface description file, which is created during code generation for the component under test. If the code interface description file is missing, the SIL/PIL

simulation cannot proceed. You see an error reporting that the file does not exist. If you select the unsupported option **Classic call interface**, this error can occur. Therefore, do not select the option.

To Workspace Block

If you enable MAT-file logging, top-model SIL/PIL and SIL/PIL blocks support To Workspace blocks.

Model block SIL/PIL does not support To Workspace blocks.

Cannot Connect SIL/PIL Outputs to Merge Block

If you connect Model block SIL/PIL or SIL/PIL block outputs to a Merge block, you see an error because S-function memory is not reusable.

Variant Condition Propagation with Variant Source and Variant Sink Blocks

Top-model SIL/PIL and SIL/PIL block simulations do not support the propagation of variant conditions across component boundaries.

Unsupported Blocks

SIL/PIL does not support the following blocks:

- Scope blocks, and all types of run-time display. For example, display of port values and signal values.
- Stop blocks. SIL/PIL ignores the Stop Simulation block and continues simulating.

Multiword Fixed-Point I/O

You cannot run SIL and PIL simulations of models that have multiword, fixed-point signals across component boundaries.

Fixed-Point Data Types Wider Than 32 Bits

SIL/PIL supports fixed-point data types that are wider than 32 bits. For example:

- 64-bit `long` and `long long`
- 64-bit execution profiling timer data type
- `int64` and `uint64` in MATLAB Coder SIL execution.

The following constraints apply:

- For 64-bit data type support, the data type must be representable as `long` or `long long` on the MATLAB host *and* the target. Otherwise, the software uses the multiword, fixed-point approach, which SIL/PIL does not support.
- The software does not support the 40-bit `long` data type of the TI's C6000™ target.

  Through the **Configuration** > **Hardware Implementation** pane, you can enable support for the 64-bit `long long` data type. For data types with widths between 33 and 40 bits (inclusive), the software implements the data types using the 40-bit `long` data type, which SIL/PIL does not support.

Data Type Replacement

The software does not support replacement data type names that you define for the built-in data type `boolean` if these names map to the `int` or `uint` built-in data type.

Continuous Sample Times

Top-model SIL/PIL and SIL/PIL block do not support continuous sample times at the SIL or PIL component boundary. However, they support continuous sample times within the component.

Model block SIL/PIL does not support continuous sample times.

Variable-Size Signals

Model block SIL/PIL simulations support variable-size signals only if **All Parameters** > **Propagate sizes of variable-size signals** is `During execution`.

Top-model SIL/PIL and SIL/PIL block simulations treat variable-size signals at the I/O boundary of the SIL/PIL component as fixed-size signals, which can lead to errors during propagation of signal sizes. To avoid such errors, use only fixed-size signals at the I/O boundary of the SIL/PIL component.

There can be cases where no error occurs during propagation of signal sizes. In these cases, the software treats variable-size input signals as zero-size signals.

Internal Signal Logging

SIL/PIL blocks do not support signal logging. For a workaround, see "Log Internal Signals of a Component" on page 64-22.

The following internal signal logging limitations apply to top-model and Model block SIL/PIL simulations.

Limitation	Applies To	
	Top-Model SIL/PIL	Model Block SIL/PIL
Only signals that are included in the C API are logged during SIL/PIL simulation. To observe the signals in the generated code, you can configure the signals as test points. For each signal, select the **Signal Properties > Test point** check box.	Yes	Yes
Signals feeding merge blocks are not supported for logging in normal simulation but are logged in SIL/PIL mode. The logged values during SIL/PIL are the same as the logged values for the output of the merge block.	Yes	No
Top-model normal simulation logs data at a periodic rate but top-model SIL/PIL simulation logs data at a constant rate under these circumstances: • **Default parameter behavior** is `Tunable`. • A constant sample time signal from a Model block is logged in the top model. • The logged signal is not directly connected to a root-level output port. To avoid this behavior and log at the constant rate in all simulation modes, set **Default parameter behavior** to `Inlined`.	Yes	No
Features not supported: • Signal logging in models referenced by the SIL/PIL component. • Signal logging in Simulink Function block. • Virtual signals, for example, mux. • Buses. • Custom storage classes.	Yes	Yes

Limitation	Applies To	
	Top-Model SIL/PIL	Model Block SIL/PIL
• Continuous, asynchronous, and triggered sample times. At the top-level of export-function models, you can log signals with triggered sample times. • Logging of Stateflow states and local data. • Units.		
Variable-size, function-call, and Action signals are not supported. A normal simulation produces an error. A SIL/PIL simulation produces a warning.	Yes	No
State port signals are not supported. A normal simulation produces an error. A SIL/PIL simulation does not produce a warning.	Yes	No

Unsupported Implementation Errors

If you use a custom storage class (CSC) with a **Type** property that is `Other` or if you use a data store, signal, or parameter implementation that SIL/PIL does not support, you can see errors like the following:

```
The following data interfaces have
implementations that are not supported by SIL or PIL.
```

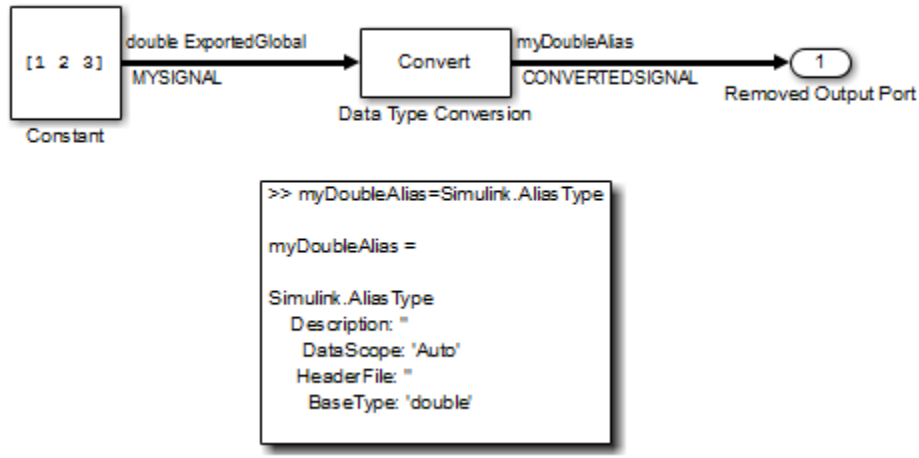
`data interfaces` can be global data stores, `inports`, `outports`, or `parameters`.

The model output port has been optimized through virtual output port optimization. See “Virtualized Output Ports Optimization” on page 55-17. The error occurs because the properties (for example, data type, dimensions) of the signal or signals entering the virtual root output port have been modified by routing the signals in one of the following ways:

- Through a Mux block.
- Through a block that changes the signal data type. To check the consistency of data types in the model, display Port Data Types by selecting **Display > Signals & Ports > Port Data Types** (see “Port Data Types” (Simulink)).
- Through a block that changes the signal dimensions. To check the consistency of data types in the model, display dimensions by selecting **Display > Signal & Ports > Signal Dimensions**.

Dimension changes from scalar (1) to matrix `[1x1]`, and, matrix `[1x1]` to scalar (1), can lead to this error. It is difficult to inspect the model for such changes because **Display > Signal & Ports > Signal Dimensions** does not distinguish between (1) and `[1x1]` dimensions. Both signals are displayed as scalar signals. Check your model and workspace objects carefully. Make sure that you specify scalar dimensions consistently.

The following model causes this error by changing the output port signal data type.



Hardware Implementation

PIL does not support multiword data types where the word order differs from the target byte order. The PIL simulation fails, displaying undefined behavior.

PIL requires that you configure the correct **Hardware Implementation** settings for the target environment, including byte ordering for targets. If you do not specify the correct byte ordering, the PIL simulation fails, displaying undefined behavior.

Non-ASCII Characters in Folder Name

If the name of the current working folder contains non-ASCII characters, you cannot run a SIL simulation.

State Logging

SIL/PIL does not support state logging (Simulink).

Bus Elements Mapped to Imported Bit-Field Definitions

If you map Simulink bus elements to bit fields through an imported header file, a SIL or PIL simulation produces a build error. For example, if your model has an Inport block connected to a bus that is a Simulink.Bus object with these properties:

- **Name** — `myBus`
- **Bus elements** — An array of Simulink.BusElement objects with these properties.

Name	DataType	Complexity	Dimensions
bitField0	boolean	real	1
bitField1	boolean	real	1
bitField2	boolean	real	1
bitField3	boolean	real	1
bitField4	boolean	real	1
bitField5	boolean	real	1

- **Data scope** — `Imported`
- **Header file** — `busSpecification.h`. This file contains `myBus`, which defines C bit-field data types for the bus elements.

```
typedef struct myBus
{
   unsigned int bitField0 : 1;
   unsigned int bitField1 : 1;
   unsigned int bitField2 : 1;
   unsigned int bitField3 : 1;
   unsigned int bitField4 : 1;
   unsigned int bitField5 : 1;
} myBus;
```

Size Mismatch Between Simulink and Target Hardware Data Types

When a Simulink data type and the corresponding target hardware data type differ in size, a SIL or PIL simulation produces an error. This size mismatch can occur if you map a Simulink data type to the target hardware data type through definitions in an imported header file. For example, if you create a data type alias, `T_BOOL`, which is a `Simulink.AliasType` object with these properties:

- **Base type** — `boolean`.

- **Mode** — `Built in`, `boolean`.
- **Data scope** — `Imported`.
- **Header file** — `myDefinitions.h`. This file defines `T_BOOL` as an enumerated data type:

```
typedef enum _BOOL_TYPE
{
  FALSE           = 0,
  TRUE            = 1
} BOOL_TYPE;

typedef BOOL_TYPE T_BOOL;
```

  In this case, the compiler for the target hardware determines the size of `T_BOOL`, which can differ from the size of the Simulink data type, `boolean`.

Top-Model SIL/PIL Limitations

Top-Model Root-Level Logging

Top-model SIL/PIL supports signal logging for signals connected to root-level inports and outports. The C API is not required. Root-level logging has the following limitations:

- The characteristics of the logged data such as data type, sample time, and dimensions must match the characteristics of the root-level inports and outports (rather than the characteristics of the connected signal).

  In some cases, there can be differences in data type and dimensions between the signal being logged and the root inport or outport that the signal is connected to. Consider the following examples.

  - If a signal being logged has matrix dimensions `[1x5]` but the outport connected to the signal has vector dimensions (`5`), then the data logged during a SIL or PIL simulation has vector dimensions (`5`).
  - If a signal being logged has scalar dimensions but the outport connected to the signal has matrix dimensions `[1x1]`, then the data logged during a SIL or PIL simulation has matrix dimensions `[1x1]`.
- Signals connected to duplicated inports are not logged during SIL/PIL simulation. No warning is issued.

  During normal simulation, signals connected directly to duplicated inports are logged.

- The Signal Logging Selector / `DataLoggingOverride` override mechanism is not supported.
- Normal and SIL/PIL simulations log bus signals with names that are different when all of the following conditions apply:
  - The `SaveOutput` or `SignalLogging` configuration parameter is `on`.
  - The names of the elements in the bus signal are different from the corresponding names in the bus object. For example, when the `InheritFromInputs` parameter for a Bus Creator block is set to `'on'`.
- The software inserts the suffix, `_wrapper` for *output logging*, if the save format is `Structure`, `Structure with time`, or `Dataset` and you run the `sim` command without specifying the single-output format. The software adds `_wrapper` to the block name for signals in `yout`. If the save format is `Array`, the software does not add the suffix. For example:

  ```
  >> yout.signals

  ans =
          values: [11x1 double]
      dimensions: 1
           label: 'SignalLogging'
       blockName: 'sillogging_wrapper/OutputLogging'
  ```

  To avoid this behavior, run command-line simulations with the `sim` command specifying the single-output format. See "Run Simulations Programmatically" (Simulink).

Model in Compiled State During Top-Model SIL/PIL

During a top-model SIL/PIL simulation, the software places the model in a compiled state – see `model`. This action can result in a conflict over global resources between the model and the generated SIL/PIL code. In this case, differences between normal mode and SIL/PIL simulation outputs can result.

For example, consider a model that uses UDP blocks from the DSP System Toolbox. These blocks open UDP sockets, which can lead to resource contention between the model and the generated SIL/PIL code.

Callback Support

SIL/PIL does not support the callbacks (model or block) `StartFcn` and `StopFcn`.

Note: Top-model SIL/PIL supports the callback `InitFcn`.

Incremental Build

When you start a top-model SIL/PIL simulation, the software regenerates code if it detects changes to your model. The software detects changes by using a checksum for the model. The software does not detect changes that you make to:

- The `HeaderFile` property of a `Simulink.AliasType` object
- Legacy S-functions

If you make these changes, build (**Ctrl-B**) your model again before starting the next PIL simulation.

Initialize, Reset, and Terminate Function Blocks

Top-model SIL/PIL supports:

- For export-function models (Simulink), Initialize Function, Reset Function, and Terminate Function blocks.
- For models that are not export-function models, Initialize Function and Terminate Function blocks.

Model Block SIL/PIL Limitations

Top-Model Code Testing

The following limitations apply:

- The `Model Variants` block does not support the block parameter `CodeInterface`. The software behaves as if `CodeInterface` is set to `'Model reference'`. To work around this limitation, use the `Variant Subsystem` block. Through this block, you can incorporate `Model` blocks for which `CodeInterface` is set to `'Top model'`.
- Because model arguments do not apply to a top model, when the **Code interface** block parameter is set to `Top model`, the software does not support the **Model arguments** block parameter.
- Conditional execution does not apply to a top model. If a Model block is set up to execute conditionally and the **Code interface** block parameter is set to `'Top model'`, the software produces an error when you run a SIL or PIL simulation.

- For sample time independent models, you must set **Configuration Parameters > Solver > Periodic sample time constraint** to `Ensure sample time independent`.
- Simulation results from top-model code and model reference code can differ when a root-level Inport is connected to a root-level Outport by a signal that has a signal object with an initial value.

  For top-model code, the software associates the signal object with the Inport. The software can apply the initial value for the signal object to the Inport. See "Initialization Behavior Summary for Signal Objects" (Simulink).

  For model reference code, the software associates the signal object with the Outport. The software does not apply the initial value for the signal object to the Inport.

Conditionally Executed Subsystem

You see an error if:

- You place your Model block, in either SIL or PIL simulation mode, in a conditionally executed subsystem and the referenced model is multirate (that is, has multiple sample times). Single-rate, referenced models (with only a single sample time) are not affected.
- Your Model block, in either SIL or PIL simulation mode, has blocks that depend on absolute time **and** is conditionally executed.

Outputs with Constant Sample Time

If the block parameter **Code interface** is `Top model`, Model block SIL/PIL supports outputs with constant sample time.

Noninlined S-Functions

Model-block SIL/PIL simulations do not support noninlined S-functions.

SIL/PIL Block Limitations

PIL Block Mux

The PIL block supports mux signals, except mixed data-type mux signals that expand into individual signals during a right-click subsystem build.

Code Coverage

SIL block simulations do not support the generation of code coverage results. PIL block support for code coverage depends on your target connectivity configuration and third-party product support.

Subsystem with Inherited Sample Time Blocks

When you create a SIL/PIL block from a subsystem that has blocks with inherited sample times, the generated code and SIL/PIL wrapper acquire the sample time of the original parent model. If you use the SIL/PIL block in a context that does not allow explicit sample times, for example, within a triggered subsystem, you see an error.

Try one of these workarounds:

- Before you create the SIL/PIL block, in the parent model, set **Configuration Parameters > Solver > Periodic sample time constraint** to `Ensure sample time independent`.
- Using the subsystem, create a Model block that is independent of sample time. With this block, run Model block SIL/PIL simulations.

Related Examples

- "SIL and PIL Simulations" on page 64-2
- "Choose a SIL or PIL Approach" on page 64-11

Check Configuration

Use the cgv.Config class to check model settings for a SIL or PIL simulation. You can review your model configuration and determine the settings that you must change. By default, `cgv.Config` changes configuration parameter values to the value that it recommends, but does not save the model. Alternatively, you can:

- Change configuration parameter values to the values that `cgv.Config` recommends, and save the model. Specify this approach using the `SaveModel` property.
- List the values that `cgv.Config` recommends for the configuration parameters, but not change the configuration parameters or the model. Specify this approach using the `ReportOnly` property.

Note:

- Execution in the target environment can require additional modifications to configuration parameter values or the model.
- Do not use referenced configuration sets in models that you are changing using `cgv.Config`. If the model uses a referenced configuration set, update the model with a copy of the configuration set. Use the `Simulink.ConfigSetRef.getRefConfigSet` method. For more information, see `Simulink.ConfigSetRef` in the Simulink documentation.
- If you use `cgv.Config` on a model that executes a callback function, the callback function can change configuration parameter values each time the model loads. The callback function can revert changes that `cgv.Config`. For more information, see "Callbacks for Customized Model Behavior" (Simulink).

To verify that your model is configured for SIL or PIL:

1. Construct a `cgv.Config` object that changes the configuration parameter values without saving the model. For example, to configure your model for SIL:

   ```
   c = cgv.Config('vdp', 'connectivity', 'sil');
   ```

   Tip:

- You can obtain a list of changes without changing the configuration parameter values. When you construct the object, include the `'ReportOnly', 'on'` property name and value pair.

- You can change the configuration parameter values and save the model. When you construct the object, include the `'SaveModel', 'on'` property name and value pair.

2 Determine and change the configuration parameter values that the object recommends using the `configModel` method. For example:

```
c.configModel();
```

3 Display a report of the changes that `configModel` makes. For example:

```
c.displayReport();
```

4 Review the changes.

5 To apply the changes to your model, save the model.

Related Examples

- "Configure and Run SIL Simulation" on page 64-15
- "Verify Numerical Equivalence with CGV" on page 64-78
- "Verify Numerical Equivalence Between Two Modes of Execution of a Model" on page 64-79

Verify Numerical Equivalence with CGV

Before verifying numerical equivalence:

- Configure your model for SIL or PIL simulation.
- Use the cgv.Config class of the CGV API to verify the model configuration for SIL or PIL simulation.
- Configure your model for code generation. For more information, see "Configure Model for Code Generation Objectives by Using Code Generation Advisor" on page 29-2.
- Save your model. If you modify a model without saving it, CGV can issue an error.

To verify numerical equivalence:

- Set up the tests for the first execution environment. For example, simulation.
- Use run (cgv.CGV) to run the tests for the first execution environment.
- Set up the tests for the second execution environment. For example, top-model PIL.
- Use `cgv.CGV.run` to run the tests for the second execution environment.
- Use getOutputData (cgv.CGV) to get the output data for each execution environment.
- Use getSavedSignals (cgv.CGV) to display the signal names in the output data. (optional)
- Build a list of signal names for input to other `cgv.CGV` methods. (optional)
- Use createToleranceFile (cgv.CGV) to create a file correlating tolerance information with output signal names. (optional)
- Use compare (cgv.CGV) to compare the output signals of the first and second execution environments for numerical equivalence.

Note: Simulink Test is a separate product that provides additional capabilities for SIL and PIL testing, for example, test sequence construction and test management.

Related Examples

- "Configure and Run SIL Simulation" on page 64-15
- "Check Configuration" on page 64-76
- "Verify Numerical Equivalence Between Two Modes of Execution of a Model" on page 64-79

Verify Numerical Equivalence Between Two Modes of Execution of a Model

In this section...

The following example describes configuring, executing, and comparing the results of the rtwdemo_cgv model in normal and software-in-the-loop (SIL) simulation modes.

Configure the Model

The first task for verifying numerical equivalence is to check the configuration of your model.

1. Open the `rtwdemo_cgv` model.

   ```
   cgvModel = 'rtwdemo_cgv';
   load_system(cgvModel);
   ```

2. Save the model to a working directory.

   ```
   save_system(cgvModel, fullfile(pwd, cgvModel));
   close_system(cgvModel); % avoid original model shadowing saved model
   ```

3. Use the cgv.Config class to create a `cgv.Config` object. Specify parameters that check and modify configuration parameter values and save the model for top-model SIL mode of execution.

   ```
   cgvCfg = cgv.Config('rtwdemo_cgv', 'connectivity', 'sil', 'SaveModel', 'on');
   ```

4. Use the configModel (cgv.Config) method to review your model configuration and to change the settings to configure your model for SIL. When `'connectivity'` is set to `'sil'`, the system target file is automatically set to `'ert.tlc'`. If you specified the parameter/value pair, (`'SaveModel', 'on'`) when you created the `cgvCfg` object, the `cgv.Config.configModel` method saves the model.

Note: CGV runs on models that are open. If you modify a model without saving it, CGV can issue an error.

```
cgvCfg.configModel(); % Evaluate, change, and save your model for SIL
```

5 Display a report of the changes that `cgv.Config.configModel` makes to the model.

```
cgvCfg.displayReport(); % In this example, this reports no changes
```

Execute the Model

Use the CGV API to execute the model in two modes. The two modes in this example are normal mode simulation and SIL mode. In each execution of the model, the CGV object for each mode captures the output data and writes the data to a file.

1 If you have not already done so, follow the steps described in "Configure the Model" on page 64-79.

2 Create a `cgv.CGV` object that specifies the `rtwdemo_cgv` model in normal mode simulation.

```
cgvSim = cgv.CGV(cgvModel, 'connectivity', 'sim');
```

Note: When the top model is set to normal simulation mode, the CGV API sets referenced models in PIL mode to accelerator mode.

3 Provide the input file to the `cgvSim` object.

```
cgvSim.addInputData(1, [cgvModel '_data']);
```

4 Before execution of the model, specify the MATLAB files to execute or MAT-files to load. This step is optional.

```
cgvSim.addPostLoadFiles({[cgvModel '_init.m']});
```

5 Specify a location where the object writes all output data and metadata files for execution. This step is optional.

```
cgvSim.setOutputDir('cgv_output');
```

6 Execute the model.

```
result1 = cgvSim.run();
```

```
*** handling PostLoad file rtwdemo_cgv_init.m
Start CGV execution of model rtwdemo_cgv, ComponentType topmodel, ...
 connectivity sim, InputData rtwdemo_cgv_data.mat
End CGV execution: status completed
```

7 Get the output data associated with the input data.

```
outputDataSim = cgvSim.getOutputData(1);
```

8 For the next mode of execution, SIL, repeat steps 2–7.

```
cgvSil = cgv.CGV( cgvModel, 'Connectivity', 'sil');
cgvSil.addInputData(1, [cgvModel '_data']);
cgvSil.addPostLoadFiles({[cgvModel '_init.m']});
cgvSil.setOutputDir('cgv_output');
result2 = cgvSil.run();
```

At the MATLAB command line, the result is:

```
*** handling PostLoad file rtwdemo_cgv_init.m
Start CGV execution of model rtwdemo_cgv, ComponentType topmodel, ...
     connectivity sil, InputData rtwdemo_cgv_data.mat

### Starting build procedure for model: rtwdemo_cgv
### Successful completion of build procedure for ...
    model: rtwdemo_cgv
### Preparing to start SIL simulation ...
### Starting SIL simulation for model: rtwdemo_cgv
### Stopping SIL simulation for model: rtwdemo_cgv
End CGV execution: status completed
```

Compare All Output Signals

After setting up and running the test, compare the outputs by doing the following:

1 If you have not already done so, configure and test the model, as described in “Configure the Model” on page 64-79 and “Execute the Model” on page 64-80.

2 Test that the execution result of the model:

```
if ~result1 || ~result2
    error('Execution of model failed.');
end
```

3 Use the getOutputData (cgv.CGV) method to get the output data from the cgv.CGV objects.

```
simData = cgvSim.getOutputData(1);
silData = cgvSil.getOutputData(1);
```

4 Display a list of signals by name using the getSavedSignals (cgv.CGV) method.

```
cgvSim.getSavedSignals(simData);
```

At the MATLAB command line, the result it:

```
simData.hi0.Data(:,1)
simData.hi0.Data(:,2)
simData.Vector.Data(:,1)
simData.Vector.Data(:,2)
simData.Vector.Data(:,3)
simData.Vector.Data(:,4)
simData.BusOutputs.hi0.Data(:,1)
simData.BusOutputs.hi0.Data(:,2)
simData.BusOutputs.hi1.mid0.lo0.Data(1,1,:)
simData.BusOutputs.hi1.mid0.lo0.Data(1,2,:)
simData.BusOutputs.hi1.mid0.lo0.Data(2,1,:)
simData.BusOutputs.hi1.mid0.lo0.Data(2,2,:)
simData.BusOutputs.hi1.mid0.lo1.Data
simData.BusOutputs.hi1.mid0.lo2.Data
simData.BusOutputs.hi1.mid1.Data(:,1)
simData.BusOutputs.hi1.mid1.Data(:,2)
simData.ErrorsInjected.Data
```

5 Using the list of signals, build a list of signals in a cell array of character vectors. The signal list can contain a number of signals.

```
signalList = {'simData.ErrorsInjected.Data'};
```

6 Use the createToleranceFile (cgv.CGV) method to create a file, in this example, `'localtol'`, correlating tolerance information with output signal names.

```
toleranceList = {{'absolute', 0.5}};
cgv.CGV.createToleranceFile('localtol', signalList, toleranceList);
```

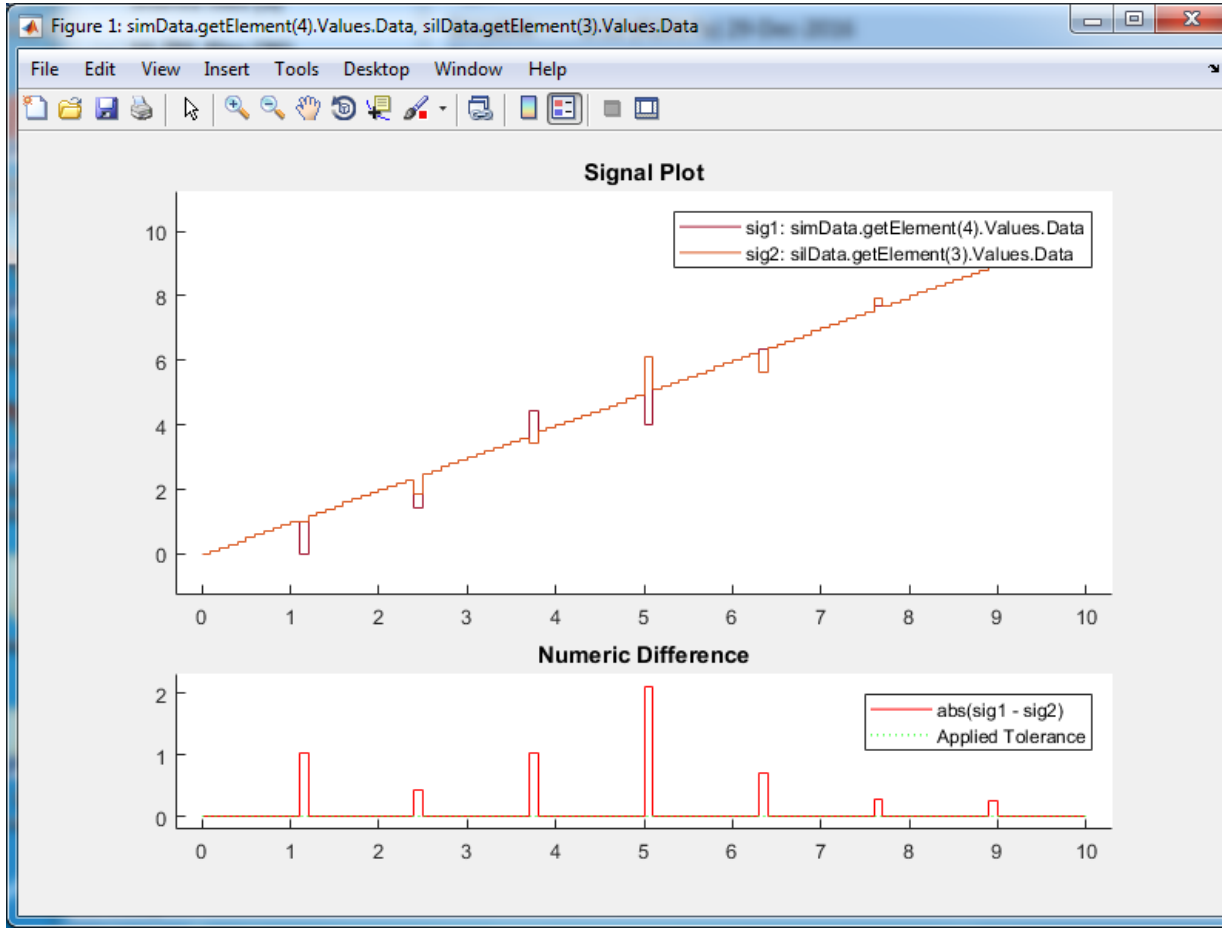
7 Compare the output data signals. By default, the compare (cgv.CGV) method looks at all signals which have a common name between both executions. If a tolerance file is present, `cgv.CGV.compare` uses the associated tolerance for a specific signal during comparison; otherwise the tolerance is zero. In this example, the `'Plot'` parameter is set to `'mismatch'`. Therefore, only mismatched signals produce a plot.

```
[matchNames, ~, mismatchNames, ~] = ...
    cgv.CGV.compare(simData, silData, 'Plot', 'mismatch', ...
    'Tolerancefile', 'localtol');
fprintf( '%d Signals match, %d Signals mismatch\n', ...
    length(matchNames), length(mismatchNames));
disp('Mismatched Signal Names:');
disp(mismatchNames);
```

At the MATLAB command line, the result is:

```
14 Signals match, 1 Signals mismatch
Mismatched Signal Names:
    'simData.ErrorsInjected.Data'
```

A plot results from the mismatch on signal `simData.ErrorsInjected.Data`.



The lower plot displays the numeric difference between the results.

Compare Individual Output Signals

After setting up and running the test, compare the outputs of individual signals by doing the following:

1 If you have not already done so, configure and test the model, as described in "Configure the Model" on page 64-79 and "Execute the Model" on page 64-80.

2 Use the getOutputData (cgv.CGV) method to get the output data from the `cgv.CGV` objects.

```
simData = cgvSim.getOutputData(1);
silData = cgvSil.getOutputData(1);
```

3 Use the getSavedSignals (cgv.CGV) method to display the output data signal names. Build a list of specific signal names in a cell array of character vectors. The signal list can contain number of signals.

```
cgv.CGV.getSavedSignals(simData);
signalList = {'simData.BusOutputs.hi1.mid0.lo1.Data', ...
'simData.BusOutputs.hi1.mid0.lo2.Data', 'simData.Vector.Data(:,3)'};
```

4 Use the specified signals as input to the compare (cgv.CGV) method to compare the signals from separate runs.

```
[matchNames, ~, mismatchNames, ~] = ...
    cgv.CGV.compare(simData, silData, 'Plot', 'mismatch', ...
    'signals', signalList);
fprintf( '%d Signals match, %d Signals mismatch\n', ...
    length(matchNames), length(mismatchNames));
if ~isempty(mismatchNames)
    disp( 'Mismatched Signal Names:');
    disp(mismatchNames);
end
```

At the MATLAB command line, the result is:

```
3 Signals match, 0 Signals mismatch
```

Plot Output Signals

After setting up and running the test, use the plot (cgv.CGV) method to plot output signals.

1 If you have not already done so, configure and test the model, as described in “Configure the Model” on page 64-79 and “Execute the Model” on page 64-80.

2 Use the getOutputData (cgv.CGV) method to get the output data from the `cgv.CGV` objects.

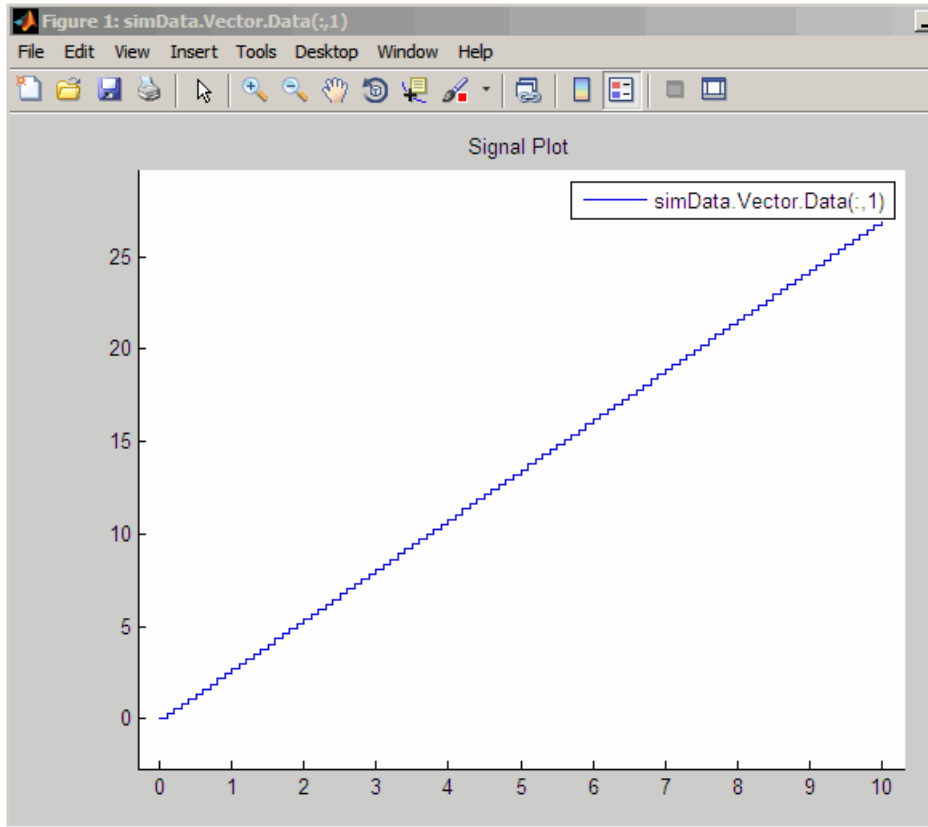
```
simData = cgvSim.getOutputData(1);
```

3 Use the getSavedSignals (cgv.CGV) method to display the output data signal names. Build a list of specific signal names in a cell array of character vectors. The signal list can contain number of signals.

```
cgv.CGV.getSavedSignals(simData);
signalList = {'simData.Vector.Data(:,1)'};
```

4 Use the specified signal list as input to the plot (cgv.CGV) method to compare the signals from separate runs.

```
[signalNames, signalFigures] = cgv.CGV.plot(simData, ...
     'Signals', signalList);
```



Related Examples

- “Verify Numerical Equivalence with CGV” on page 64-78
- “Check Configuration” on page 64-76

Using Code Generation Verification API

Configure and run normal, software-in-the-loop (SIL), and processor-in-the-loop (PIL) simulations, and compare results.

Note: Simulink Test is a separate product that provides additional capabilities for SIL and PIL testing, for example, test sequence construction and test management.

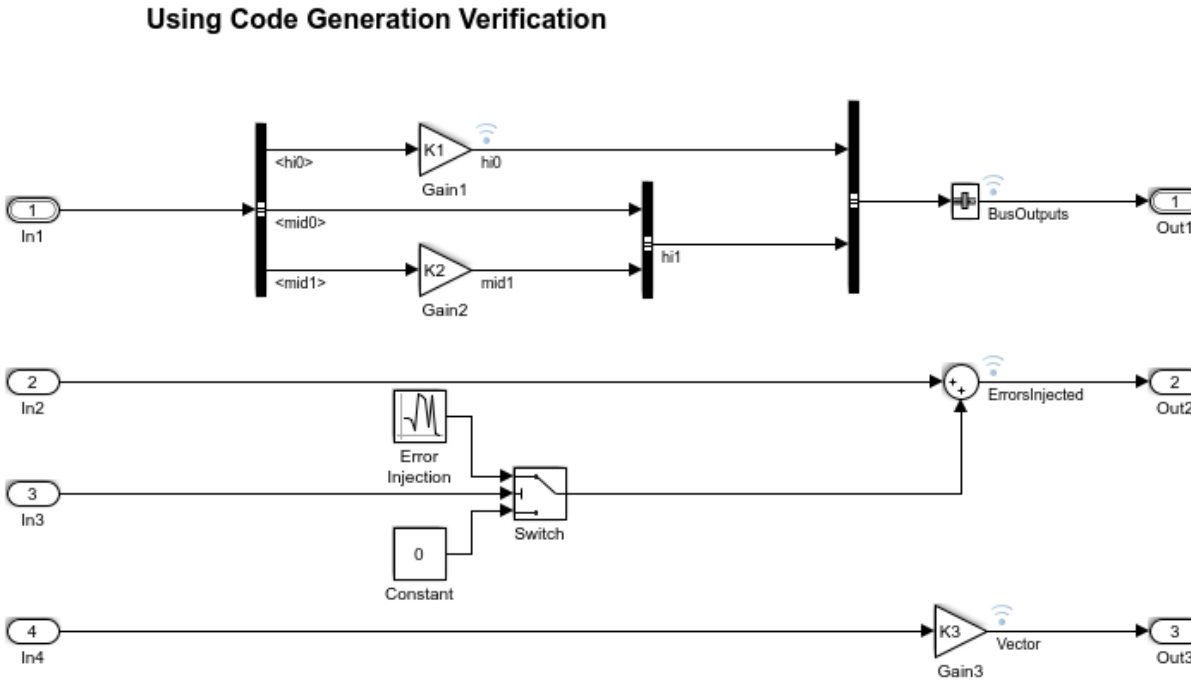
Review the Model

The rtwdemo_cgv model uses buses, scalars, and vectorized data, plus error injection to create differences between test executions.

Note: Before executing the code in this example, change to a writable folder. If you are not working in a writable folder, code generation errors occur.

To open rtwdemo_cgv, in the MATLAB® Command Window, enter the following commands.

```
baseVars = who;  % For future cleanup.
cgvModel = 'rtwdemo_cgv';
close_system(cgvModel,0);
open_system( cgvModel);
```



The model contains a hierarchical bus with three nested buses. This arrangement of buses produces complex hierarchical data at the first logged outport. At the second outport, the model injects errors in the signal at fixed intervals. These errors produce different results between two runs. The signal at the third outport is a vector of four values per sample to help show the comparison support.

Verify the Model Configuration

CGV provides a class, cgv.Config, to check whether models have a configuration that is compatible with execution in a SIL or PIL environment using an ert target. This model has already been modified using the cgv.Config class.

Execute Under CGV

Run in Normal and SIL Modes

The model executes in three modes under CGV: normal, SIL, and PIL. In each case, the CGV object captures the output data and writes it to a file. For more information, see CGV Documentation. To execute the model in normal and SIL simulation modes, enter:

```
cgvSim = cgv.CGV( cgvModel, 'Connectivity', 'Normal');
cgvSim.addInputData(1, [cgvModel '_data']);
% This next CGV function, addPostLoadFiles(), allows you to specify MATLAB(R)
% programs to execute, or mat-files to load, before execution of the model.
cgvSim.addPostLoadFiles({[cgvModel '_init.m']});
cgvSim.setOutputDir('cgv_output');
cgvSim.activateConfigSet( 'CS1_default');
result1 = cgvSim.run();

% CGV provides methods to simplify numerical equivalence checking.
% The copySetup method creates an exact duplicate of an existing CGV object without
% results data.  You can change the SimulationMode using setMode() and then
% execute again.
cgvSil = cgvSim.copySetup();
cgvSil.setMode( 'SIL');
% You can provide a baseline file to CGV for comparing the simulation
% output. In this example, the comparison results set the status to
% 'failed', because the ErrorsInjected signal differs between simulations.
cgvSil.addBaseline( 1, 'rtwdemo_cgv_results');
result2 = cgvSil.run();

% To see the name(s) of the signal(s) that did match, use getMismatches.
% Mismatched signal names are only available if a baseline was added and
% the comparison failed.
if strcmp( cgvSil.getStatus( 1), 'failed')
    disp( 'Mismatched Signal Names:');
    [signalNames, plotFiles] = cgvSil.getMismatches( 1);
    fprintf( 1, 'Signal Names: %s\n', signalNames{:} );
    fprintf( 1, 'Path to plot files: %s\n', plotFiles{:});
    assert(numel(signalNames)==1,'Expected exactly one mismatch');
end
```

```
Applying Configuration Set:
  CS1_default
Applying PostLoad file:
  B:\matlab\toolbox\rtw\rtwdemos\rtwdemo_cgv_init.m
Starting execution:
```

```
  ComponentType: topmodel
  Connectivity: normal
  InputData:
  B:\matlab\toolbox\rtw\rtwdemos\rtwdemo_cgv_data.mat
End CGV execution: status completed.
Applying PostLoad file:
  B:\matlab\toolbox\rtw\rtwdemos\rtwdemo_cgv_init.m
Starting execution:
  ComponentType: topmodel
  Connectivity: sil
  InputData:
  B:\matlab\toolbox\rtw\rtwdemos\rtwdemo_cgv_data.mat
### Starting build procedure for model: rtwdemo_cgv
### Successful completion of build procedure for model: rtwdemo_cgv
### Preparing to start SIL simulation ...
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
### Updating code generation report with SIL files ...
### Starting SIL simulation for component: rtwdemo_cgv
### Stopping SIL simulation for component: rtwdemo_cgv
End CGV execution: status failed.
Mismatched Signal Names:
Signal Names: simout.getElement(3).Values.Data
Path to plot files: C:\TEMP\Bdoc17a_538369_5692\IB_CPU_21\tp32af565d\ex96023632\cgv_out
```

Run in PIL Mode

Next, the model runs a PIL simulation, using your embedded processor. A universal embedded processor does not exist. Therefore, PIL support is provided by using the host computer where MATLAB® is running. The host processor is treated as an embedded target.

A customization file is executed that maps this model's PIL execution onto the SIL infrastructure. After the customization file is executed, CGV execution displays PIL messages for the mode. SIL messages display the connectivity target.

The configuration set for the model is already configured with: **Hardware Implementation > Test hardware > Test hardware is the same as production hardware** is checked. **Code Generation > Verification > Enable portable word sizes** is checked. These settings work in SIL and in PIL when PIL is mapped onto the SIL connectivity target.

```
copyfile( which( 'rtwdemo_cgv_sl_customization.m'), fullfile( pwd, 'sl_customization.m
```

```
sl_refresh_customizations();

cgvPil = cgvSim.copySetup();
cgvPil.setMode( 'PIL');
result3 = cgvPil.run();

Applying PostLoad file:
  B:\matlab\toolbox\rtw\rtwdemos\rtwdemo_cgv_init.m
Starting execution:
  ComponentType: topmodel
  Connectivity: pil
  InputData:
  B:\matlab\toolbox\rtw\rtwdemos\rtwdemo_cgv_data.mat
### Starting build procedure for model: rtwdemo_cgv
### Successful completion of build procedure for model: rtwdemo_cgv
### Preparing to start PIL simulation ...
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
### Connectivity configuration for "C:\TEMP\Bdoc17a_538369_5692\IB_CPU_21\tp32af565d\ex
### Updating code generation report with PIL files ...
### Starting application: 'rtwdemo_cgv_ert_rtw\pil\rtwdemo_cgv.exe'
End CGV execution: status completed.
```

Remove Customization

To prevent problems with other models, immediately remove the customization used to show PIL mode.

```
delete( 'sl_customization.m');
sl_refresh_customizations();
```

Check that execution did not terminate with an error

The run() function returns a Boolean value, which is true if the execution completes without model compilation or simulation error. Before accessing the data, check the result returned from each execution.

```
if ~result1 || ~result2 || ~result3
    disp('Execution of model failed.');
end

simData   = cgvSim.getOutputData(1);
silData   = cgvSil.getOutputData(1);
pilData   = cgvPil.getOutputData(1);
```

Compare Results

The executions are now complete. Compare the results. The comparison code supports a plot with filters. Plots display both the data and the difference.

CGV functions display signals names (as used in the command window) and create a file correlating tolerance information with signal names.

Show Signal Names from Normal Simulation

Display a list of signal names from the saved data.

Note: cgv.CGV.compare ignores signals that appear in only one data set. For example, the compare function ignores a logged internal signal hi0 that appears in the output of a normal simulation, but does not appear in the output of a SIL simulation.

```
cgv.CGV.getSavedSignals( simData);
```

```
simData.getElement(1).Values.Data(:,1)
simData.getElement(1).Values.Data(:,2)
simData.getElement(2).Values.Data(:,1)
simData.getElement(2).Values.Data(:,2)
simData.getElement(2).Values.Data(:,3)
simData.getElement(2).Values.Data(:,4)
simData.getElement(3).Values.hi0.Data(:,1)
simData.getElement(3).Values.hi0.Data(:,2)
simData.getElement(3).Values.hi1.mid0.lo0.Data(1,1,:)
simData.getElement(3).Values.hi1.mid0.lo0.Data(2,1,:)
simData.getElement(3).Values.hi1.mid0.lo0.Data(1,2,:)
simData.getElement(3).Values.hi1.mid0.lo0.Data(2,2,:)
simData.getElement(3).Values.hi1.mid0.lo1.Data
simData.getElement(3).Values.hi1.mid0.lo2.Data
simData.getElement(3).Values.hi1.mid1.Data(:,1)
simData.getElement(3).Values.hi1.mid1.Data(:,2)
simData.getElement(4).Values.Data
```

Create a Tolerance File

The CGV createToleranceFile function creates a file correlating tolerance information with signal names. For the options available to configure tolerances, see cgv.CGV.createToleranceFile. By default, tolerances are zero. Therefore the signals must match exactly. This example allows a delta of 0.5 on the ErrorsInjected signal.

```
signalList = {'simData.ErrorsInjected.Data' };
```

```
toleranceList = { { 'absolute', 0.5}};
cgv.CGV.createToleranceFile( 'localtol', signalList, toleranceList );
```

Compare Signals

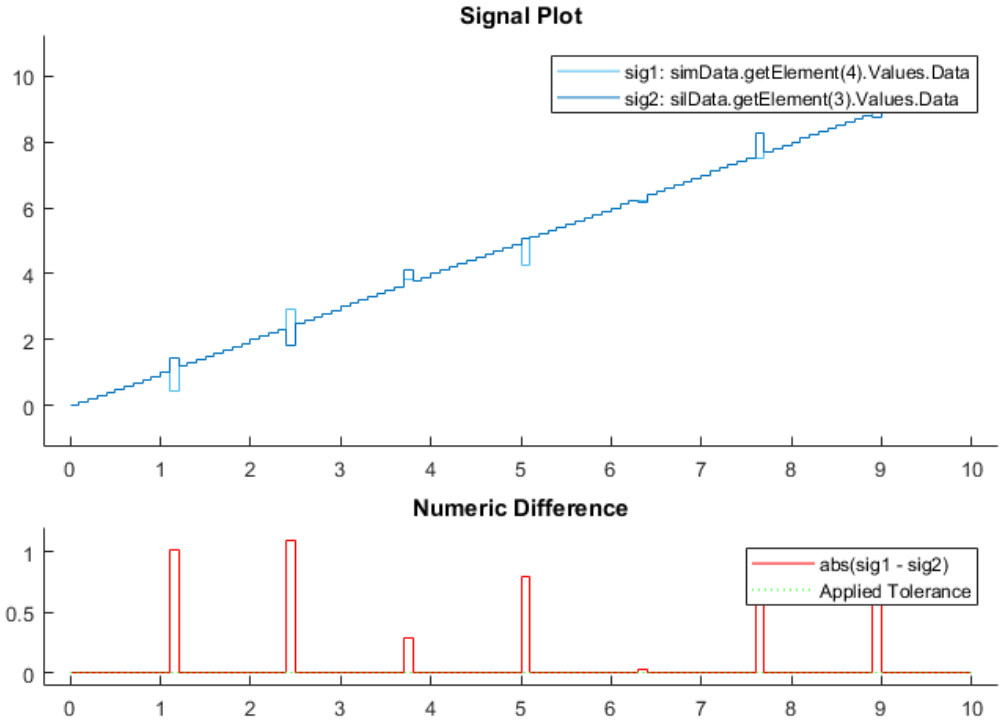
By default, the cgv.CGV.compare function looks at signals that have a common name between both executions. In the following code, the simData.hi0.Data signals are not compared, because the signals do not appear in silData.

The second and fourth return parameters of the compare function are for matched figures and mismatched figures. Tildes (~) represent these parameters because this example does not use the return values.

A plot results from the mismatch on signal simData.ErrorsInjected.Data.

```
[matchNames, ~, mismatchNames, ~] = ...
    cgv.CGV.compare( simData, silData, 'Plot', 'mismatch', ...
    'Tolerancefile', 'localtol');
fprintf( '%d Signals match, %d Signals mismatch\n', ...
    length(matchNames), length(mismatchNames));
assert(length(mismatchNames) == 1,'Expected exactly one mismatch');
assert(length(matchNames) == 14, 'Expected exactly 14 matches');

disp( 'Mismatched Signal Names:');
disp(mismatchNames');
```

```
14 Signals match, 1 Signals mismatch
Mismatched Signal Names:
    'simData.getElement(4).Values.Data'
```



Compare Individual Signals

The cgv.CGV.compare function also compares only the specified signals. In this example, the function compares only three signals.

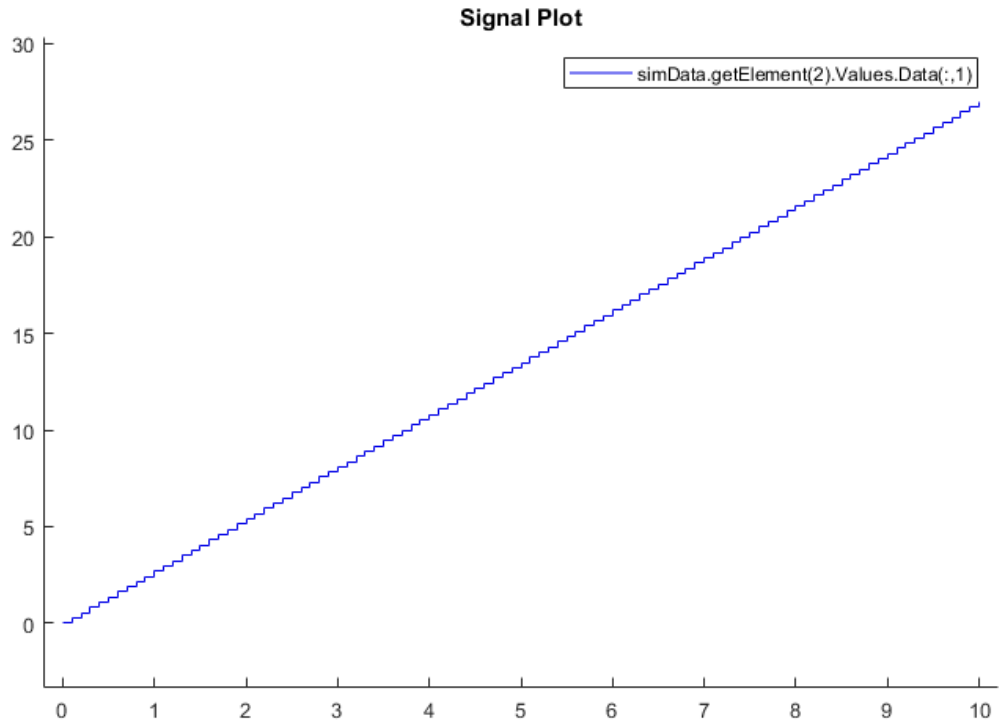
```
[matchNames, ~, mismatchNames, ~ ] = ...
    cgv.CGV.compare( simData, silData, 'Plot', 'mismatch', ...
    'Signals', {'simData.getElement(3).Values.hi1.mid0.lo1.Data', 'simData.getElement(3
    'simData.getElement(2).Values.Data(:,3)'});
fprintf( '%d Signals match, %d Signals mismatch\n', ...
    length(matchNames), length(mismatchNames));
assert(isempty(mismatchNames),'Expected no mismatches');
if ~isempty(mismatchNames)
    disp( 'Mismatched Signal Names:');
    disp(mismatchNames');
end
```

```
% Since a mismatch does not occur for these signals, a plot is not generated.

3 Signals match, 0 Signals mismatch
```

Additional Plotting Support

To create a plot of a list of signals, call cgv.CGV.plot. For example,

```
[signalNames, signalFigures] = cgv.CGV.plot( simData, ...
    'Signals', {'simData.getElement(2).Values.Data(:,1)'});
```



View Signal Data in the Simulation Data Inspector Tool

To open the Simulation Data Inspector tool, at the MATLAB® command line, enter Simulink.sdi.view. To import the signal data, in the Simulation Data Inspector tool,

select **File > Import Data**, which opens the Data Import tool. Select **Import from > Base workspace**, to view and select the signals saved in simData and silData.

Clear Your Workspace

Clear the variables from the workspace:

```
newBaseVars = who;
addedVars = setdiff( newBaseVars, baseVars);
clearCmd = ['clear ' sprintf( '%s ', addedVars{:})];
eval( clearCmd);
clear newBaseVars addedVars clearCmd
rtwdemoclean;
```

Related Examples

- "Simulink Test"
- "Verify Numerical Equivalence with CGV" on page 64-78
- "Verify Numerical Equivalence Between Two Modes of Execution of a Model" on page 64-79

Numerical Consistency between Model and Generated Code

Numerical Consistency of Model and Generated Code Simulation Results

In this section...

Numerical Consistency

In the Model-Based Design workflow (Simulink Code Inspector), you use MathWorks products to generate code for numerical applications that employ fixed-point and floating-point arithmetic.

- To develop models, you use MATLAB, Simulink, and Stateflow.
- To generate source code, you use Simulink Coder and Embedded Coder.
- To test numerical equivalence between your model and generated code, you compare model and generated code simulation results. For example, normal mode simulation results compared with software-in-the-loop (SIL) simulation results.

The results from the model and generated code simulations are numerically consistent if:

- In fixed-point applications, the results agree in a bit-wise comparison.
- In floating-point applications, the results agree with an error tolerance that you specify.

Use the Simulation Data Inspector to compare results. To determine whether discrepancies exist or are significant, you can specify absolute and relative tolerance values:

- For fixed-point applications, you can specify an absolute tolerance of zero.
- For floating-point applications, you can specify tolerance with respect to a reference value or signal. The choice of reference depends on your application. Consider these examples:
  - An algorithm that solves a linear algebraic equation by iterative, feed-forward error calculations. You can specify tolerance with respect to `eps`.

- A Proportional-Integral-Derivative (PID) controller for a closed-loop system. For transient behavior, you can specify tolerance with criteria from a standard. For steady-state behavior, you can specify tolerance with reference to the PID controller characteristics.

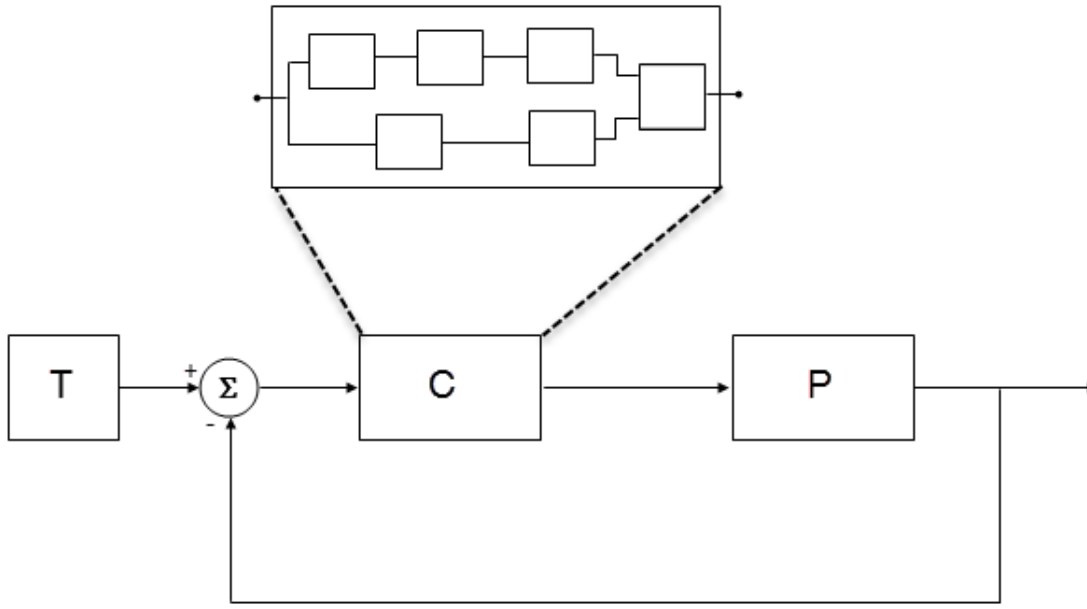
Programmatically, you can specify absolute and relative tolerance values through the `absTol` and `relTol` properties of the `Simulink.sdi.Signal` object.

Numerical Consistency in Complex Systems

For complex systems, numerical differences between model and generated code simulations can be a result of block-level differences propagating through the system. If you observe numerical differences at the system level:

1. Identify blocks for which block-level numerical differences exceed the error tolerance.
2. Investigate each identified block.

Consider the following plant-controller model.



- T produces reference or test signals.
- C is the controller component. The controller output is the plant input. C can be a Model block that comprises multiple Model blocks.
- P is the plant component. The plant output is subtracted from the reference signal to produce the controller input.

To test numerical equivalence between the model controller and the generated code version:

1. Run the model in normal mode, and, using the Simulation Data Inspector, record the output of C.
2. Specify SIL mode for C. Rerun the simulation, recording the output of C.
3. Using the Simulation Data Inspector, compare normal and SIL mode outputs with reference to your specified error tolerance.

If the Simulation Data Inspector comparison indicates a match, the model and generated code results are numerically consistent.

If the normal and SIL mode outputs do not match:

1. Within C, enable signal logging for block outputs.
2. Run the model in normal mode.
3. Rerun the simulation with C in SIL mode.
4. Using the Simulation Data Inspector, compare the logged output signals with reference to your specified error tolerance. See "Compare Simulation Data" (Simulink).
5. Identify blocks for which normal and SIL mode output differences exceed the error tolerance.
6. Analyze each identified block and look for the cause. For example, the generated code might use a different math library than MATLAB.

Note: If the comparison of a large number of signals is required, you can automate the workflow with Simulink Test. See "Code Generation Verification Workflow with Simulink Test" (Simulink Test).

Reasons for Block-Level Numerical Differences

In fixed-point and floating-point application development, there are factors that can affect numerical agreement between block-level results from model and generated code simulations.

Some factors can affect both fixed-point and floating-point applications. For example, the use of:

- Code generation optimization.
- Custom code.
- Code replacement library entries whose results differ from MATLAB results.
- Code replacement libraries that implement different algorithms.

Other factors affect only floating-point applications. For example:

- Numerical soundness of algorithm.

- Algorithm sensitivity to input.
- Closed loop and open loop behavior.

References

[1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003, pp. 15–17.

See Also

`eps`

Related Examples

- “Record Data with the Simulation Data Inspector” (Simulink)
- “Compare Simulation Data” (Simulink)
- “Inspect and Compare Data Programmatically” (Simulink)
- “Code Generation Verification Workflow with Simulink Test” (Simulink Test)

More About

- “How the Simulation Data Inspector Compares Data” (Simulink)
- “Differences Between Generated Code and MATLAB Code” (Simulink)
- “Code Replacement”
- “Types of In-the-Loop Testing in the V-Model”
- MATLAB Function (Simulink)
- “SIL and PIL Limitations” on page 64-61

Software-in-the-Loop Execution for MATLAB Coder

Code Verification Through Software-in-the-Loop and Processor-in-the-Loop Execution

MATLAB Coder supports software-in-the-loop (SIL) and processor-in-the-loop (PIL) execution, which enables you to verify production-ready source code and compiled object code. With these execution modes, you can reuse test vectors developed for your MATLAB functions to verify the numerical behavior of library code.

In SIL execution, through a MATLAB SIL interface, the software compiles and runs library code on your development computer. In PIL execution, through a MATLAB PIL interface, the software cross-compiles and runs production object code on a target processor or an equivalent instruction set simulator. Before you run a PIL execution, you must set up a PIL connectivity configuration for your target.

The workflow for generating and verifying code is:

1. Set up MATLAB Coder.
2. Fix errors detected at design time.
3. Generate MEX function.
4. Test MEX function.
5. Generate C/C++ library code.
6. Verify generated C/C++ code through SIL or PIL execution — requires Embedded Coder license.

In step 4, you verify code that is generated for execution within MATLAB. However, this code is different from the standalone code generated for libraries. In step 6, with an Embedded Coder license, you use SIL or PIL execution to verify the standalone code.

For more information, use the following table.

Feature	See
SIL execution	• "Software-in-the-Loop Execution with the MATLAB Coder App" on page 66-4 • "Software-in-the-Loop Execution From Command Line" on page 66-6

Feature	See
PIL target connectivity configuration	• "Create PIL Target Connectivity Configuration" on page 66-12 • "Host-Target Communication for PIL" on page 66-16 • "Processor-in-the-Loop Execution From Command Line" on page 66-27
PIL execution	• "Processor-in-the-Loop Execution with the MATLAB Coder App" on page 66-25 • "Processor-in-the-Loop Execution From Command Line" on page 66-27
Code generation, MEX functions, and libraries	• "MATLAB Code Analysis" (MATLAB Coder) • "Generating Code" (MATLAB Coder) • "Deployment" (MATLAB Coder)

Software-in-the-Loop Execution with the MATLAB Coder App

Use software-in-the-loop (SIL) execution to verify the numerical behavior of the generated C/C++ code with reference to your original MATLAB functions.

1. To open the MATLAB Coder app, on the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the app icon.
2. To open your project, click  , and then click `Open existing project`. Select the project. For example, `kalman_filter01.prj`.
3. On the **Generate Code** page, click the **Generate** arrow  .
4. In the **Generate** dialog box:
   a. Set **Build type** to `Static Library` or `Dynamic Library`.
   b. In the **Output file name** field, use the default value. For example, `kalman01`.
   c. Specify **Language**.
   d. Clear the **Generate code only** check box.
   e. In the **Hardware Board** field, use the default value (`MATLAB Host Computer`).

   You do not have to specify the **Toolchain** setting. By default, the MATLAB Coder app locates an installed toolchain.
5. To generate the C or C++ code, click **Generate**.
6. Click **Verify Code**.
7. In the command field, specify the test file that calls the original MATLAB functions, for example, `test01_ui.m`.
8. If required, select the **Enable source-level debugging for SIL** check box.
9. To start the SIL execution, click **Run Generated Code**.

   The MATLAB Coder app:

   - Generates a standalone library, for example, `codegen\lib\kalman01`.
   - Generates SIL interface code, for example, `codegen\lib\kalman01\sil`.
   - Runs the test file, replacing calls to the MATLAB function with calls to the generated code in the library.

- Displays messages from the SIL execution in the **Test Output** tab.

10 Verify that the results from the SIL execution match the results from the original MATLAB functions.

11 To terminate the SIL execution process, click **Stop SIL Verification**. Alternatively, on the **Test Output** tab, click the link that follows `To terminate execution`.

Note: On a Windows operating system, the Windows Firewall can potentially block the SIL execution. To allow the SIL execution, use the Windows Security Alert dialog box. For example, in Windows 7, click **Allow access**.

Related Examples

- "C Code Generation Using the MATLAB Coder App" (MATLAB Coder)
- "Software-in-the-Loop Execution From Command Line" on page 66-6
- "Debug Generated Code During SIL Execution" on page 66-9
- "Generate Execution Time Profile" on page 59-3

More About

- "Code Verification Through Software-in-the-Loop and Processor-in-the-Loop Execution" on page 66-2

Software-in-the-Loop Execution From Command Line

Use software-in-the-loop (SIL) execution to verify the numerical behavior of the generated C/C++ code with reference to your original MATLAB functions.

To set up and start a SIL execution from the command line:

1. Create a coder.EmbeddedCodeConfig object.
2. Configure the object for SIL.
3. Use the `codegen` function to generate library code for your MATLAB function and the SIL interface.
4. Use the `coder.runTest` function to run the test file for your original MATLAB function.

To terminate the SIL execution, use the `clear` *`function`*`_sil` or `clear mex` command.

The following example shows how you can set up and run a SIL execution from the command line.

SIL Execution of Code Generated for a Kalman Estimator

1 Copy MATLAB code for Kalman estimator

From *`docroot`*`\toolbox\coder\examples\kalman`, copy the following files to your working folder:

- `kalman01.m` — MATLAB function for the Kalman estimator
- `test01_ui.m` — MATLAB file to test `kalman01.m`
- `plot_trajectory.m` — File that plots actual target trajectory and Kalman estimator output
- `position.mat` — Input data

```
src_dir = ...
    fullfile(docroot,'toolbox','coder','examples','kalman');

copyfile(fullfile(src_dir,'kalman01.m'), '.')
copyfile(fullfile(src_dir,'test01_ui.m'), '.')
```

```
copyfile(fullfile(src_dir,'plot_trajectory.m'), '.')
copyfile(fullfile(src_dir,'position.mat'), '.')
```

For a description of the Kalman estimator in this example, see "C Code Generation at the Command Line" (MATLAB Coder).

2 Configure SIL execution

a From your working folder, create a coder.EmbeddedCodeConfig object.

```
config = coder.config('lib');
config.GenerateReport = true; % Optional, documents code in HTML report
```

b Configure the object for SIL.

```
config.VerificationMode = 'SIL';

% Check that production hardware setting is the default
% i.e. 'Generic->MATLAB Host Computer'
disp(config.HardwareImplementation.ProdHWDeviceType);
```

c If required, enable the Microsoft Visual Studio debugger for SIL execution:

```
config.SILDebugging = true;
```

3 Generate code and run SIL execution

a Generate library code for the `kalman01` MATLAB function and the SIL interface.

```
codegen('-config', config, '-args', {zeros(2,1)}, 'kalman01');
```

The software creates the following output folders:

- `codegen\lib\kalman01` — Standalone code for `kalman01`.
- `codegen\lib\kalman01\sil` — SIL interface code for `kalman01`.

b Run the MATLAB test file `test01_ui` with `kalman01_sil`. `kalman01_sil` is the SIL interface for `kalman01`.

```
coder.runTest('test01_ui', ['kalman01_sil.' mexext]);
```

Verify that the output of this run matches the output from the original `kalman01.m` function.

Note: On a Windows operating system, the Windows Firewall can potentially block the SIL execution. To allow the SIL execution, use the Windows Security Alert dialog box. For example, in Windows 7, click **Allow access**.

4 Debug code during SIL execution

If you enable the Microsoft Visual Studio debugger, then running the test file opens the Microsoft Visual Studio IDE with debugger breakpoints at the start of the `kalman01_initialize` and `kalman01` functions.

You can use the debugger features to observe code behavior. For example, you can step through code and examine variables.

To end the debugging session:

a Remove all breakpoints.

b Click the `Continue` button (**F5**).

The SIL execution runs to completion.

5 Terminate SIL execution

Terminate the SIL execution process.

```
clear kalman01_sil;
```

You can also use the command `clear mex`, which clears MEX functions from memory.

Related Examples

- "C Code Generation Using the MATLAB Coder App" (MATLAB Coder)
- "Software-in-the-Loop Execution with the MATLAB Coder App" on page 66-4
- "Debug Generated Code During SIL Execution" on page 66-9
- "Generate Execution Time Profile" on page 59-3

More About

- "Code Verification Through Software-in-the-Loop and Processor-in-the-Loop Execution" on page 66-2

Debug Generated Code During SIL Execution

If a SIL execution fails or you notice differences between the outputs of your original functions and the generated code, you can rerun the SIL execution with a debugger enabled. By inserting breakpoints, you can observe the behavior of code sections, which might help you to understand the cause of the problem.

For a SIL execution failure, you can also view information from the standard output and standard error streams in the MATLAB Command Window. For example:

- Output from `printf` statements in your code.
- If you enable run-time error detection, messages sent to `stderr`.
- Some low-level system messages.

Note: During a SIL execution, the SIL application redirects the `stdout` and `stderr` streams. When the application terminates, the MATLAB Command Window displays the information from the redirected streams. The SIL application also provides a basic signal handler, which captures the POSIX signals `SIGFPE`, `SIGILL`, `SIGABRT`, and `SIGSEV`. For this signal handler, the SIL application includes the file `signal.h`.

A SIL execution supports the following debuggers:

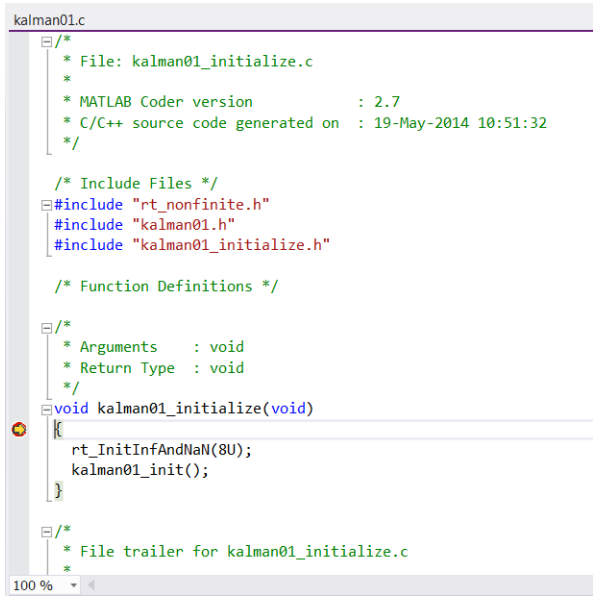
- On Windows, Microsoft Visual Studio debugger.
- On Linux, GNU Data Display Debugger (DDD).

Note: You can perform SIL debugging only if your Microsoft Visual C++ or GNU GCC compiler is supported by the MATLAB product family. For more information, see supported compilers.

To run a SIL execution with debugging enabled:

1. On the **Generate Code** page, click **Verify Code**.
2. Select the **Enable source-level debugging for SIL** check box.
3. Click **Run Generated Code**.

On a Windows computer, your *`user_fn`*`.c` or *`user_fn`*`.cpp` file opens in the Microsoft Visual Studio IDE with debugger breakpoints at the start of the *`user_fn`*`_initialize` and *`user_fn`* functions.



You can now use the debugger features to observe code behavior. For example, you can step through code and examine variables.

To end the debugging session:

1. Remove all breakpoints.
2. Click the `Continue` button (**F5**).

   The SIL execution runs to completion.
3. To terminate the SIL execution process, on the **Test Output** tab, click the link that follows `To terminate execution`, for example, `clear kalman01_sil`.

   The Microsoft Visual Studio IDE closes automatically.

Note: If you select **Debug** > **Stop Debugging**, the SIL execution times out with the following error message:

```
Communications error: failed to send data to the target. There might be
multiple reasons for this failure.

...
...
```

Related Examples

- “Software-in-the-Loop Execution with the MATLAB Coder App” on page 66-4
- “Software-in-the-Loop Execution From Command Line” on page 66-6

Create PIL Target Connectivity Configuration

In this section...

Target Connectivity Configurations for PIL

Use target connectivity configurations and the target connectivity API to customize processor-in-the-loop (PIL) execution for your target environments.

Through a target connectivity configuration, you specify:

- A target connectivity configuration name for a target connectivity API implementation.
- Settings that define compatible MATLAB code. For example, the code that is generated for a particular hardware implementation.

A PIL execution requires a target connectivity PIL API implementation that integrates third-party tools for:

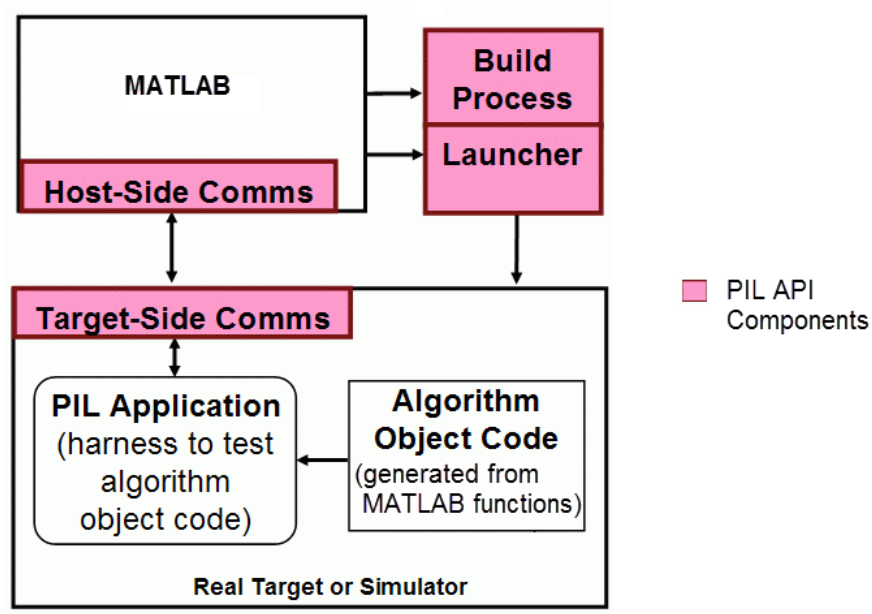
- Building the PIL application that runs on the target hardware
- Downloading, starting, and stopping the application on the target
- Communicating between MATLAB and the target

You can have many different connectivity configurations for PIL execution. Register a connectivity configuration with MATLAB by creating an `rtwTargetInfo.m` file and placing it on the MATLAB search path.

In a PIL execution, the software determines which of the available connectivity configurations to use. The software looks for a connectivity configuration that is compatible with the MATLAB code under test. If the software finds multiple or no compatible connectivity configurations, the software generates an error message with information about resolving the problem.

Create a Target Connectivity API Implementation

This diagram shows the components of the PIL target connectivity API.



You must provide implementations of the three API components:

- Build API — Specify a toolchain approach for building generated code.
- Launcher API — Control how MATLAB starts and stops the PIL executable.
- Communications API — Customize connectivity between MATLAB and the PIL target. Embedded Coder provides host-side support for TCP/IP and serial communications, which you can adapt for other protocols.

These steps outline how you create a target connectivity API implementation. The example code shown in the steps is taken from the `ConnectivityConfig.m` file used in "Processor-in-the-Loop Execution From Command Line" on page 66-27.

1 Create a subclass of `rtw.connectivity.Config`.

```
ConnectivityConfig < rtw.connectivity.Config
```

2 In the subclass:

- Instantiate `rtw.connectivity.MakefileBuilder`, which configures the build process.

```
builder = rtw.connectivity.MakefileBuilder(componentArgs, ...
                targetApplicationFramework, ...
                exeExtension);
```

- Create a subclass of `rtw.connectivity.Launcher`, which downloads and executes the application using a third-party tool.

```
launcher = mypil.Launcher(componentArgs, builder);
```

3 Configure your `rtiostream` API implementation of the host-target communications on page 66-16 channel.

- For the target side, you must provide the driver code for communications, for example, code for TCP/IP or serial communications. To integrate this code into the build process, create a subclass of `rtw.pil.RtIOStreamApplicationFramework`.
- For the host side, you can use a supplied library for TCP/IP or serial communications. Instantiate `rtw.connectivity.RtIOStreamHostCommunicator`, which loads and initializes the library that you specify.

```
hostCommunicator = rtw.connectivity.RtIOStreamHostCommunicator(...
                        componentArgs, ...
                        launcher, ...
                        rtiostreamLib);
```

4 If you require execution-time profiling of generated code, create a timer object that provides details of the hardware-specific timer and associated source files. See "Specify Hardware Timer" on page 66-22.

Register Target Connectivity Configuration

To register a target connectivity API implementation as a target connectivity configuration in MATLAB:

1 Create or update an `rtwTargetInfo.m` file. In this file:

- Create a target connectivity configuration object that specifies, for example, the configuration name for a target connectivity API implementation and compatible MATLAB code.
- Invoke `registerTargetInfo`.

2 Add the folder containing `rtwTargetInfo.m` to the search path and refresh the MATLAB Coder library registration information.

For more information, see `rtw.connectivity.ConfigRegistry`.

Verify Target Connectivity Configuration

To verify your target connectivity configuration early on and independently of your algorithm development and code generation, use the `piltest` function. With the function, you can run a suite of tests. The function:

- Runs the MATLAB function and performs PIL executions.
- Compares results and produces errors if it detects differences.

For an example, see "PIL Execution of Code Generated for a Kalman Estimator" on page 66-27.

See Also

`piltest` | `rtw.connectivity.Config` | `rtw.connectivity.ConfigRegistry` | `rtw.connectivity.Launcher` | `rtw.connectivity.MakefileBuilder` | `rtw.connectivity.RtIOStreamHostCommunicator` | `rtw.pil.RtIOStreamApplicationFramework`

Related Examples

- "Processor-in-the-Loop Execution From Command Line" on page 66-27
- "Subclass Constructors" (MATLAB)
- "Host-Target Communication for PIL" on page 66-16
- "Specify Hardware Timer" on page 66-22

Host-Target Communication for PIL

In this section...

Communications `rtiostream` API

The `rtiostream` API supports communications for the target connectivity API. Use the `rtiostream` API to implement a communication channel that enables data exchange between different processes.

PIL verification requires a host-target communications channel. This communications channel comprises driver code that runs on the host and target. The `rtiostream` API defines the signature of target-side and host-side functions that must be implemented by this driver code.

The API is independent of the physical layer that sends the data. Possible physical layers include RS232, Ethernet, or Controller Area Network (CAN).

A full `rtiostream` implementation requires both host-side and target-side drivers. Code generation software includes host-side drivers for the default TCP/IP implementation as well as a version for serial communications. To use:

- The TCP/IP `rtiostream` communications channel, you must provide, or obtain from a third party, target-specific TCP/IP device drivers.
- The serial communications channel, you must provide, or obtain from a third party, target-specific serial device drivers.

For other communication channels and platforms, the code generation software does not provide default implementations. You must provide both the host-side and the target-side drivers.

The `rtiostream` API comprises the following functions:

- `rtIOStreamOpen`

- `rtIOStreamSend`
- `rtIOStreamRecv`
- `rtIOStreamClose`

For information about:

- Using `rtiostream` functions in a connectivity implementation, see "Create a Target Connectivity API Implementation" on page 66-13.
- Testing the `rtiostream` shared library methods from MATLAB code, see `rtiostream_wrapper`.
- Debugging and verifying the behavior of custom `rtiostream` interface implementations, see "Test an `rtiostream` Driver" on page 66-18.

Synchronize Host and Target

If you use the `rtiostream` API to implement the communications channel, the host and target must be synchronized, which prevents MATLAB from transmitting and receiving data before the target application is fully initialized.

To synchronize the host and target for TCP/IP rtiostream implementations, use the `setInitCommsTimeout` method from `rtw.connectivity.RtIOStreamHostCommunicator` . This approach works well for connection-oriented TCP/IP `rtiostream` implementations because MATLAB automatically waits until the target server is running.

With other `rtiostream` implementations, for example, serial, the MATLAB side of the `rtiostream` connection opens without waiting for the target to be fully initialized. In this case, you must make your `Launcher` implementation wait until the target application is fully initialized. Use one of the following approaches to synchronize your host and target:

- Add a pause at the end of the `Launcher` implementation that makes the `Launcher` wait until target initialization is complete.
- In the `Launcher` implementation, use third-party downloader or debugger APIs that wait until target initialization is complete.
- Implement a handshaking mechanism in the `Launcher` / `rtiostream` implementation that confirms completion of target initialization.

Test an `rtiostream` Driver

Use a test suite to debug and verify the behavior of custom `rtiostream` interface implementations.

The test suite has the following advantages:

- Reduces time for integrating custom hardware that does not have built-in `rtiostream` support.
- Reduces time for testing custom `rtiostream` drivers.
- Helps analyze the performance of custom `rtiostream` drivers.

The test suite has two parts. One part of the test suite runs on the target.

Note: After building the target application, download it to the target and run it.

To start this part, compile and link the following files, which are in the folder *`matlabroot`*`/toolbox/coder/rtiostream/src/rtiostreamtest` (open).

- `rtiostreamtest.c`
- `rtiostreamtest.h`
- `rtiostream.h`, located in the folder *`matlabroot`*`/rtw/c/src` (open)
- `rtiostream` implementation under investigation (for example, `rtiostream_tcpip.c`)
- `main.c`

To run the MATLAB part of the test suite, invoke `rtiostreamtest`. The syntax is as follows:

```
rtiostreamtest(connection,param1,param2)
```

- `connection` is a character vector indicating the communication method. It can have values `'tcp'` or `'serial'`.
- `param1` and `param2` have different values depending on the value of `connection`.
  - If `connection` is `'tcp'`, then `param1` and `param2` are hostname and port, respectively. For example, `rtiostreamtest('tcp', 'localhost', 2345)`.

- If `connection` is `'serial'`, then `param1` and `param2` are COM port and baud rate, respectively. For example, `rtiostreamtest('serial', 'COM1', 9600)`.

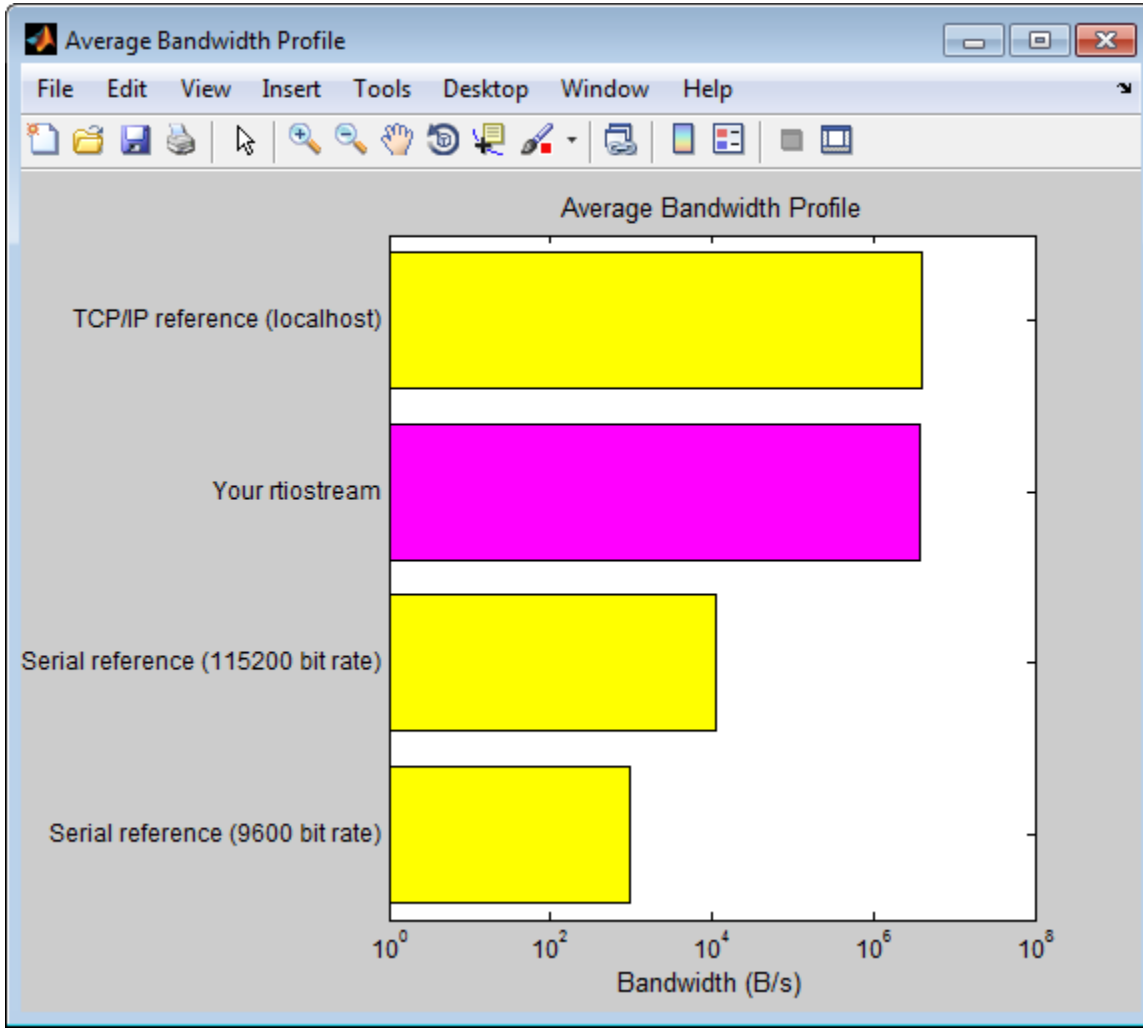
You can run the MATLAB part of the test suite as follows:

```
rtiostreamtest('tcp','localhost','2345')
```

An output in the following format appears in the MATLAB window:

```
### Test suite for rtiostream ###
Initializing connection with target...

### Hardware characteristics discovered
Size of char    : 8 bit
Size of short   : 16 bit
Size of int     : 32 bit
Size of long    : 32 bit
Size of float   : 32 bit
Size of double  : 64 bit
Size of pointer : 64 bit
Byte ordering   : Little Endian

### rtiostream characteristics discovered
Round trip time : 0.96689 ms
rtIOStreamRecv behavior : non-blocking

### Test results
Test 1 (fixed size data exchange): ......... PASS
Test 2 (varying size data exchange): ......... PASS

### Test suite for rtiostream finished successfully ###
```

Furthermore, the following profile appears.



See Also

`rtiostream_wrapper` | `rtIOStreamClose` | `rtIOStreamOpen` | `rtIOStreamRecv` | `rtIOStreamSend` | `rtw.connectivity.RtIOStreamHostCommunicator`

Related Examples

- “Create PIL Target Connectivity Configuration” on page 66-12

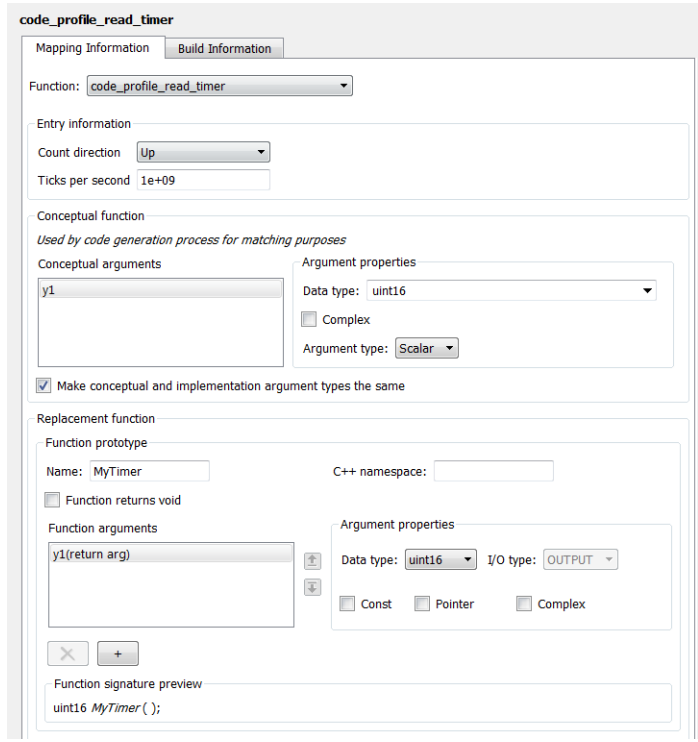
Specify Hardware Timer

For processor-in-the-loop (PIL) code execution profiling, you must create a timer object that provides details of the hardware-specific timer and associated source files. You can use the Code Replacement Tool or the code replacement library API to specify this hardware-specific timer.

To specify the timer with the Code Replacement Tool:

1. Open the Code Replacement Tool. In the Command Window, enter `crtool`.
2. Create a new code replacement table. Select **File** > **New table**.
3. Create a new function entry. Under **Tables List**, right-click the new table. Then, from the context-menu, select **New entry** > **Function**.
4. In the middle view, select the new unnamed function.
5. On the **Mapping Information** pane:
   a. From the **Function** drop-down list, select `code_profile_read_timer`.
   b. Specify the count direction for your timer. For example, from the **Count direction** drop-down list, select `Up`.
   c. In the **Ticks per second** field, specify the number of ticks per second for your timer, for example, `1e+09`.

      The default value is 0. In this case, the software reports time measurements in terms of ticks, not seconds.
   d. In the **Name** field, specify a replacement function name, for example, `MyTimer`.
   e. Click **Apply**.



f To validate the function entry, click **Validate entry**.

6 On the **Build Information** pane, specify the required build information. See "Specify Build Information for Replacement Code" on page 51-59.

7 Save the table (**Ctrl+S**). When you save the table for the first time, use the Save As dialog box to specify the file name and location.

You must save the table in a location that is on the MATLAB search path. For example, you can save this file in the folder for your subclass of `rtw.connectivity.Config`.

The software stores your timer information as a code replacement library table.

8 Assuming you save the table as *`MyCrlTable.m`*, in your subclass of `rtw.connectivity.Config`, add the following line:

```
setTimer(this, MyCrlTable)
```

Related Examples

- "Create a Target Connectivity API Implementation" on page 66-13
- "Generate Execution Time Profile" on page 59-3
- "Specify Build Information for Replacement Code" on page 51-59

More About

- "What Is Code Replacement?" on page 38-2
- "What Is Code Replacement Customization?" on page 51-3

Processor-in-the-Loop Execution with the MATLAB Coder App

Use processor-in-the-loop (PIL) execution to verify the numerical behavior of cross-compiled object code with reference to your original MATLAB functions.

Before you run a PIL execution, you must define a target connectivity configuration. In "Processor-in-the-Loop Execution From Command Line" on page 66-27, steps 1 and 2 of the example PIL Execution of Code Generated for a Kalman Estimator show how you can set up and register a connectivity configuration for PIL execution on your development computer.

1. To open the MATLAB Coder app, on the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the app icon.
2. To open your project, click  , and then click `Open existing project`. Select the project. For example, `kalman_filter.prj`.
3. On the **Generate Code** page, click the **Generate** arrow  .
4. In the **Generate** dialog box:
   a. Set **Build type** to `Static Library` or `Dynamic Library`.
   b. In the **Output file name** field, use the default value. For example, `kalman01`.
   c. Clear the **Generate code only** check box.
   d. From the **Hardware Board** drop-down list, select `None - Select device below`.
   e. In the **Device** fields, specify vendor and type. These settings must match the target hardware settings in the `rtwTargetInfo.m` file of your target connectivity configuration. For host-based PIL, select settings that match your host computer. For example:
      - For a Windows 64-bit system, set **Device vendor** to `Intel` and **Device type** to `x86-64 (Windows64)`. In addition, set **Enable long long** to `Yes`.
      - For a Linux 64-bit system, set **Device vendor** to `Intel` and **Device type** to `x86-64 (Linux 64)`.
      - For a Mac OS X system, set **Device vendor** to `Intel` and **Device type** to `x86-64 (Mac OS X)`.

You do not have to specify the **Toolchain** setting. By default, the MATLAB Coder app locates an installed toolchain.

5. To generate the C or C++ code, click **Generate**.
6. Click **Verify Code**.
7. In the command field, specify the test file that calls the original MATLAB functions, for example, `test01_ui.m`.
8. To start the PIL execution, click **Run Generated Code**.

   The MATLAB Coder app:

   - Generates a standalone library, for example, `codegen\lib\kalman01`.
   - Generates PIL interface code, for example, `codegen\lib\kalman01\pil`.
   - Runs the test file, replacing calls to the MATLAB function with calls to the generated code in the library.
   - Displays messages from the PIL execution in the **Test Output** tab.
9. Verify that the results from the PIL execution match the results from the original MATLAB functions.
10. To terminate the PIL execution process, click **Stop PIL Verification**. Alternatively, on the **Test Output** tab, click the link that follows `To terminate execution`.

Related Examples

- "C Code Generation Using the MATLAB Coder App" (MATLAB Coder)
- "Processor-in-the-Loop Execution From Command Line" on page 66-27
- "Generate Execution Time Profile" on page 59-3

More About

- "Code Verification Through Software-in-the-Loop and Processor-in-the-Loop Execution" on page 66-2

Processor-in-the-Loop Execution From Command Line

Use processor-in-the-loop (PIL) execution to verify code that you intend to deploy in production.

To set up and start a PIL execution from the command line:

1. Create, register, and verify your target connectivity configuration.
2. Create a coder.EmbeddedCodeConfig object.
3. Configure the object for PIL.
4. Use the `codegen` function to generate library code for your MATLAB function and the PIL interface.
5. Use the `coder.runTest` function to run the test file for your original MATLAB function.

To terminate the PIL execution, use the `clear` *`function`*`_pil` or `clear mex` command.

The following example shows how you can use line commands to set up and run a PIL execution on your development computer.

PIL Execution of Code Generated for a Kalman Estimator

1 Create a target connectivity API implementation

a. In your current working folder, make a local copy of the connectivity classes.

```
src_dir = ...
    fullfile(matlabroot,'toolbox','coder','simulinkcoder','+coder','+mypil');
if exist(fullfile('.','+mypil'),'dir')
    rmdir('+mypil','s')
end
mkdir +mypil
copyfile(fullfile(src_dir,'Launcher.m'), '+mypil');
copyfile(fullfile(src_dir,'TargetApplicationFramework.m'), '+mypil');
copyfile(fullfile(src_dir,'ConnectivityConfig.m'), '+mypil');
```

b. Make the copied files writable.

```
fileattrib(fullfile('+mypil', '*'),'+w');
```

c. Update the package name to reflect the new location of the files.

```
coder.mypil.Utils.UpdateClassName(...
    './+mypil/ConnectivityConfig.m',...
    'coder.mypil',...
    'mypil');
```

d Check that you now have a folder `+mypil` in the current folder, which includes three files, `Launcher.m`, `TargetApplicationFramework.m`, and `ConnectivityConfig.m`.

```
dir './+mypil'
```

e Review the code that starts the PIL application. The `mypil.Launcher` class configures a tool for starting the PIL executable. Open this class in the editor.

```
edit(which('mypil.Launcher'))
```

Review the content of this file. For example, consider the `setArgString` method. This method allows additional command line parameters to be supplied to the application. These parameters can include a TCP/IP port number. For an embedded processor implementation, you might have to hard code these settings.

f The class `mypil.ConnectivityConfig` configures target connectivity.

```
edit(which('mypil.ConnectivityConfig'))
```

Review the content of this file. For example:

- The creation of an instance of `rtw.connectivity.RtIOStreamHostCommunicator` that configures the host side of the TCP/IP communications channel.
- A call to the `setArgString` method of `Launcher` that configures the target side of the TCP/IP communications channel.
- A call to `setTimer` that configures a timer for execution time measurement. To define your own target-specific timer for execution time profiling, you must use the Code Replacement Library to specify a replacement for the function `code_profile_read_timer`.

g Review the target-side communication drivers.

```
rtiostreamtcpip_dir=fullfile(matlabroot,'rtw','c','src','rtiostream',...
    'rtiostreamtcpip');
edit(fullfile(rtiostreamtcpip_dir,'rtiostream_tcpip.c'))
```

Scroll down to the end of this file. The file contains a TCP/IP implementation of the functions `rtIOStreamOpen`, `rtIOStreamSend`, and`rtIOStreamRecv`.

These functions are required for target communication with the host. For each of these functions, you must provide an implementation that is specific to your target hardware and communication channel.

The `mypil.TargetApplicationFramework` class adds target-side communication drivers to the connectivity configuration.

```
edit(which('mypil.TargetApplicationFramework'))
```
The file specifies additional files to include in the build.

2 Register a target connectivity configuration

Use an `rtwTargetInfo.m` file to:

- Create a target connectivity configuration object.
- Invoke `registerTargetInfo`, which registers the target connectivity configuration.

The target connectivity configuration object specifies, for example:

- The configuration name and associated API implementation. See `rtw.connectivity.ConfigRegistry`
- A toolchain for your target hardware. This example assumes that the target hardware is your host computer, and uses the toolchain supplied for host-based PIL verification. For information about toolchains, see "Custom Toolchain Registration" (MATLAB Coder).

a Insert the following code into your `rtwTargetInfo.m` file, and save the file in the current working folder or in a folder that is on the MATLAB search path:

```
function rtwTargetInfo(tr)
% Register PIL connectivity config: mypil.ConnectivityConfig

tr.registerTargetInfo(@loc_createConfig);

% local function
function config = loc_createConfig

% Create object for connectivity configuration
config = rtw.connectivity.ConfigRegistry;
% Assign connectivity configuration name
config.ConfigName = 'My PIL Example';
% Associate the connectivity configuration with the connectivity
```

```
% API implementation
config.ConfigClass = 'mypil.ConnectivityConfig';

% Specify toolchains for host-based PIL
config.Toolchain =  rtw.connectivity.Utils.getHostToolchainNames;

% Through the HardwareBoard and TargetHWDeviceType properties,
% define compatible code for the target connectivity configuration
config.HardwareBoard = {};  % Any hardware board
config.TargetHWDeviceType = {'Generic->32-bit x86 compatible' ...
                             'Generic->Custom' ...
                             'Intel->x86-64 (Windows64)', ...
                             'Intel->x86-64 (Mac OS X)', ...
                             'Intel->x86-64 (Linux 64)'};
```

b Refresh the MATLAB Coder library registration information.

```
RTW.TargetRegistry.getInstance('reset');
```

3 Verify target connectivity configuration

Use the supplied `piltest` function to verify your target connectivity configuration.

a Create a `coder.EmbeddedCodeConfig` object for verifying the target connectivity configuration.

```
configVerify = coder.config('lib');
```

b Specify the manufacturer and test hardware type. For example, PIL execution on a 64-bit Windows development computer requires:

```
configVerify.HardwareImplementation.TargetHWDeviceType =...
   'Intel->x86-64 (Windows64)';
configVerify.HardwareImplementation.ProdLongLongMode = true;
```

c Run `piltest`.

```
piltest(configVerify, 'ConfigParam', {'ProdLongLongMode'} )
```

4 Copy MATLAB code for Kalman estimator

Copy the MATLAB code to your working folder.

```
src_dir = ...
    fullfile(docroot,'toolbox','coder','examples','kalman');
```

```
copyfile(fullfile(src_dir,'kalman01.m'), '.')
copyfile(fullfile(src_dir,'test01_ui.m'), '.')
copyfile(fullfile(src_dir,'plot_trajectory.m'), '.')
copyfile(fullfile(src_dir,'position.mat'), '.')
```

For a description of the Kalman estimator in this example, see "C Code Generation at the Command Line" (MATLAB Coder).

5 Configure the PIL execution

a Create a `coder.EmbeddedCodeConfig` object.

```
config = coder.config('lib');
```

b Configure the object for PIL.

```
config.VerificationMode = 'PIL';
```

c Specify production hardware, which must match one of the test hardware settings in `rtwTargetInfo.m`. For PIL execution on your development computer, specify settings that match the computer. For example, if your computer is a Windows 64-bit system, specify:

```
config.HardwareImplementation.ProdHWDeviceType =...
   'Intel->x86-64 (Windows64)';
config.HardwareImplementation.ProdLongLongMode = true;
```

For a Linux 64-bit system, set `ProdHWDeviceType` to `'Intel->x86-64 (Linux 64)'`.

For a Mac OS X system, set `ProdHWDeviceType` to `'Intel->x86-64 (Mac OS X)'`.

6 Generate code and run PIL execution

a Generate library code for the `kalman01` MATLAB function and the PIL interface.

```
codegen('-config', config, '-args', {zeros(2,1)}, 'kalman01');
```

The software creates the following output folders:

- `codegen\lib\kalman01` — Standalone code for `kalman01`.
- `codegen\lib\kalman01\pil` — PIL interface code for `kalman01`.

b Run the MATLAB test file `test01_ui` with `kalman01_pil`. `kalman01_pil` is the PIL interface for `kalman01`.

```
coder.runTest('test01_ui', ['kalman01_pil.' mexext]);
```

Verify that the output of this run matches the output from the original `kalman01.m` function.

7 Terminate PIL execution

Terminate the PIL execution process.

```
clear kalman01_pil;
```

Related Examples

- "C Code Generation Using the MATLAB Coder App" (MATLAB Coder)
- "Processor-in-the-Loop Execution with the MATLAB Coder App" on page 66-25
- "Generate Execution Time Profile" on page 59-3

More About

- "Code Verification Through Software-in-the-Loop and Processor-in-the-Loop Execution" on page 66-2

Verification of Code Generation Assumptions

The settings on the **More Settings > Hardware** tab specify target behavior, which result in the implementation of implicit assumptions in the generated code. Incorrect settings can lead to:

- Suboptimal code
- Code execution failure, incorrect code output, and nondeterministic code behavior

At the start of a processor-in-the-loop (PIL) execution, the software verifies the **Hardware** tab settings with reference to the target hardware. The software checks:

- The correctness of settings. For example, the integer bit length in the **Sizes > int** field.
- Whether the settings are optimized. For example, the rounding of signed integer division in the **Signed integer division rounds to** field.

If required, the software generates warnings and errors.

Related Examples

- “Processor-in-the-Loop Execution with the MATLAB Coder App” on page 66-25

SIL/PIL Execution Support and Limitations

Feature		Supported
Output types	Static library	Yes
	Dynamic library	Yes
	Executable	No
Languages	C	Yes
	C++	Yes
Interface types	Inputs	Yes
	Outputs	Yes
	Constant inputs	Yes
	Global data	Yes. SIL and PIL execution supports four types of storage classes on page 77-2 for MATLAB Coder global variables. The synchronization (MATLAB Coder) of global data between MATLAB and the SIL or PIL application depends on the type of storage class that you specify: • `ExportedGlobal` (default) — The synchronization of global data between MATLAB and a SIL or PIL application is identical to the synchronization between MATLAB and a MEX function. • `ExportedDefine` — There is no synchronization of global data between MATLAB and the SIL or PIL application. The application uses the values of the global variables in MATLAB at the time of code generation. • `ImportedExtern` and `ImportedExternPointer` — There is no synchronization of

Feature		Supported
		global data between MATLAB and the SIL or PIL application. The application uses initial values of global variables, which you specify in the external code. If the global variables are not initialized in the external code, the SIL or PIL execution results are undefined.
	Constant global data	Yes
	Reentrant code	Yes
	Multiple entry points	Yes
Data types	Basic types	Yes
	Enumerated types	Yes
	Structures	Yes
	Complex data	Yes
	Fixed-point data	Yes
	Multiword fixed-point data	SIL only
	`char` arrays	Yes
	Empty values	Yes
	Cell arrays	Yes
Size	Scalars	Yes
	Fixed-size arrays	Yes
	Static variable-size arrays	Yes
	Dynamic variable-size size arrays	Yes

Related Examples

- “Code Verification Through Software-in-the-Loop and Processor-in-the-Loop Execution” on page 66-2

Code Coverage in Embedded Coder

Simulink Code Coverage Metrics

In this section...

If you have a Simulink Verification and Validation license, you can run a SIL or PIL simulation that produces code coverage metrics for generated model code. The simulation performs several types of code coverage analysis.

Statement Coverage for Code Coverage

Statement coverage determines the number of source code statements that execute when the code runs. Use this type of coverage to determine whether every statement in the program has been invoked at least once.

Statement coverage = (Number of executed statements / Total number of statements) *100

Statement Coverage Example

This code snippet contains five statements. To achieve 100% statement coverage, you need at least three test cases. Specifically, tests with positive *x* values, negative *x* values, and *x* values of zero.

```
if (x > 0)
  printf( "x is positive" );
else if (x < 0)
  printf( "x is negative" );
else
  printf( "x is 0" );
```

Condition Coverage for Code Coverage

Condition coverage analyzes statements that include conditions in source code. Conditions are C/C++ Boolean expressions that contain relation operators (<, >, <=, or >=), equation operators (!= or ==), or logical negation operators (!), but that do not contain logical operators (&& or ||). This type of coverage determines whether every condition has been evaluated to all possible outcomes at least once.

Condition coverage = (Number of executed condition outcomes / Total number of condition outcomes) *100

Condition Coverage Example

In this expression:

```
y = x<=5 && x!=7;
```

there are these conditions:

```
x<=5
x!=7
```

Decision Coverage for Code Coverage

Decision coverage analyzes statements that represent decisions in source code. Decisions are Boolean expressions composed of conditions and one or more of the logical C/C++ operators && or ||. Conditions within branching constructs (if/else, while, do-while) are decisions. Decision coverage determines the percentage of the total number of decision outcomes the code exercises during execution. Use this type of coverage to determine whether all decisions, including branches, in your code are tested.

Note: The decision coverage definition for DO-178C compliance differs from the Simulink Verification and Validation definition. For decision coverage compliance with DO-178C, select the `Condition Decision` structural coverage level for Boolean expressions not containing && or || operators.

Decision coverage = (Number of executed decision outcomes / Total number of decision outcomes) *100

Decision Coverage Example

This code snippet contains three decisions:

```
y = x<=5 && x!=7;    // decision #1

if( x > 0 )     // decision #2
  printf( "decision #2 is true" );
else if( x < 0 && y )    // decision #3
  printf( "decision #3 is true" );
else
  printf( "decisions #2 and #3 are false" );
```

Modified Condition/Decision Coverage (MCDC) for Code Coverage

Modified condition/decision coverage (MCDC) is the extent to which the conditions within decisions are independently exercised during code execution.

- All conditions within decisions have been evaluated to all possible outcomes at least once.
- Every condition within a decision independently affects the outcome of the decision.

MCDC coverage = (Number of conditions evaluated to all possible outcomes affecting the outcome of the decision / Total number of conditions within the decisions) *100

Modified Condition/Decision Coverage Example

For this decision:

```
X || ( Y && Z )
```

the following set of test cases delivers 100% MCDC coverage.

	X	Y	Z
Test case #1	0	0	1
Test case #2	0	1	0
Test case #3	0	1	1
Test case #4	1	0	1

Cyclomatic Complexity for Code Coverage

Cyclomatic complexity is a measure of the structural complexity of code that uses the McCabe complexity measure. To compute the cyclomatic complexity of code, code coverage uses this formula:

$$c = \sum_{1}^{N} (o_n - 1)$$

N is the number of decisions in the code. o_n is the number of outcomes for the n^{th} decision point. Code coverage adds 1 to the complexity number for each C/C++ function.

Coverage Example

For this code snippet:

```
void evalNum( int x ){

 if (x > 0)
   printf( "x is positive" );
 else if (x < 0)
   printf( "x is negative" );
 else
  printf( "x is 0" );
}
```

the cyclomatic complexity is 3.

Relational Boundary for Code Coverage

Relational boundary code coverage examines code that has relational operations. Relational boundary code coverage metrics align with those for model coverage, as described in "Relational Boundary Coverage" (Simulink Verification and Validation). Fixed-point values in your model are integers during code coverage.

Related Examples

- "Code Coverage for Models in Software-in-the-Loop (SIL) Mode and Processor-in-the-Loop (PIL) Mode" on page 67-6

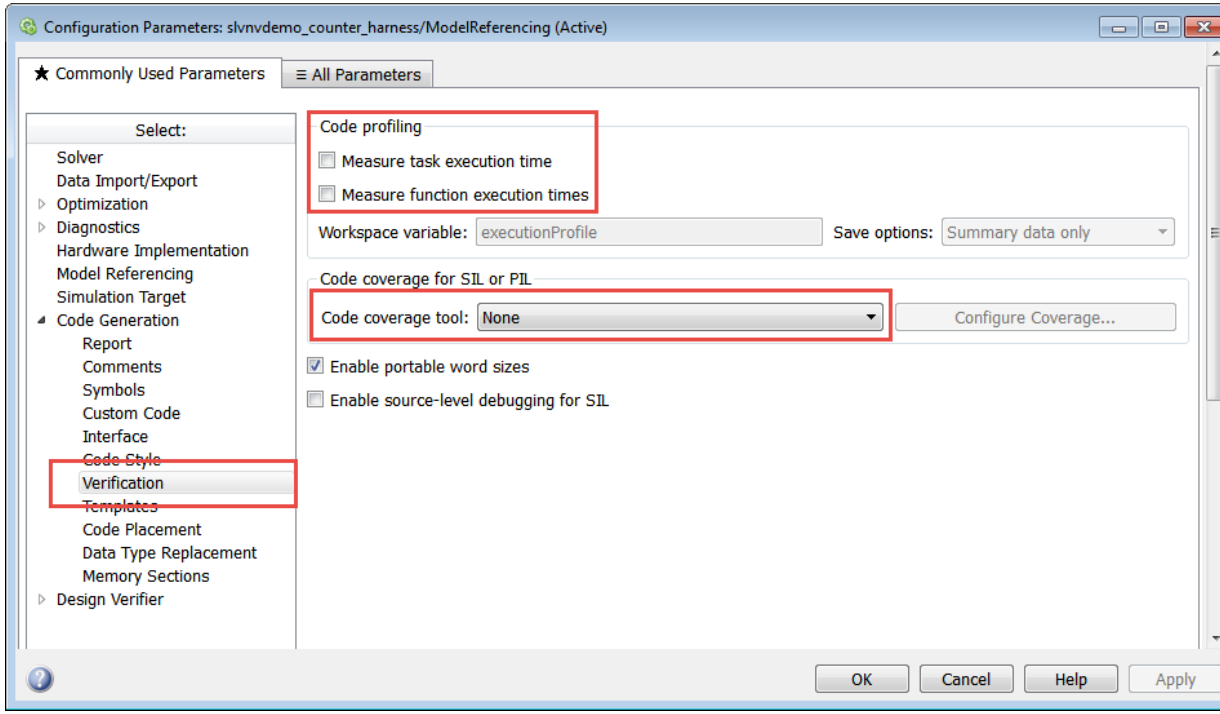
Code Coverage for Models in Software-in-the-Loop (SIL) Mode and Processor-in-the-Loop (PIL) Mode

In this section...

Requirements to Enable SIL or PIL Code Coverage for a Model

If you have both an Embedded Coder license and a Simulink Verification and Validation license, you can measure coverage for code generated from models in software-in-the-loop (SIL) mode or processor-in-the-loop (PIL) mode. The following configurations must apply for the parameters in **Code Generation > Verification**:

- Under **Code profiling**, clear **Measure function execution times**.
- Under **Code coverage for SIL or PIL**, the selected **Code coverage tool** must be `None`.



Conditions for Simulink Verification and Validation Code Coverage Measurement

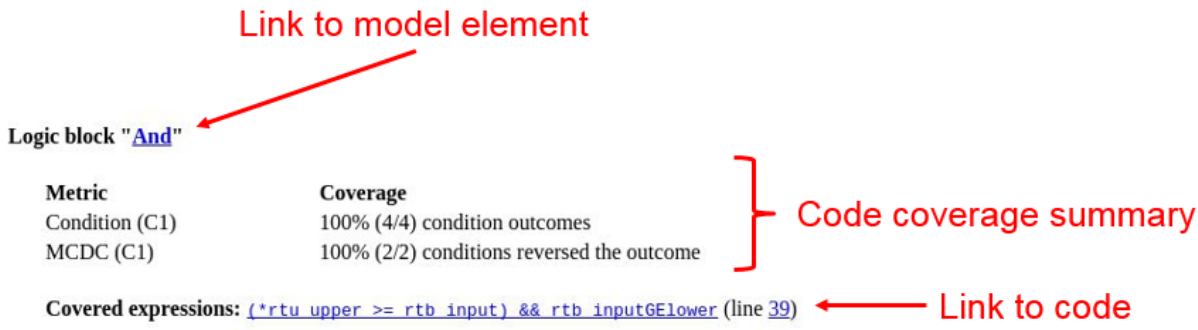
There are two workflows for measuring code coverage:

- The top model is in SIL mode or PIL mode. Measures code coverage for the top model depending on `RecordCoverage`. Also measures code coverage for referenced models, depending on `CovModelRefEnable`.
- The top model is in Normal mode and contains at least one reference model in SIL or PIL mode. Measures code coverage for the referenced model if `CovModelRefEnable` is `'on'`, `'all'`, or `'filtered'` and `RecordCoverage` is `'off'`.

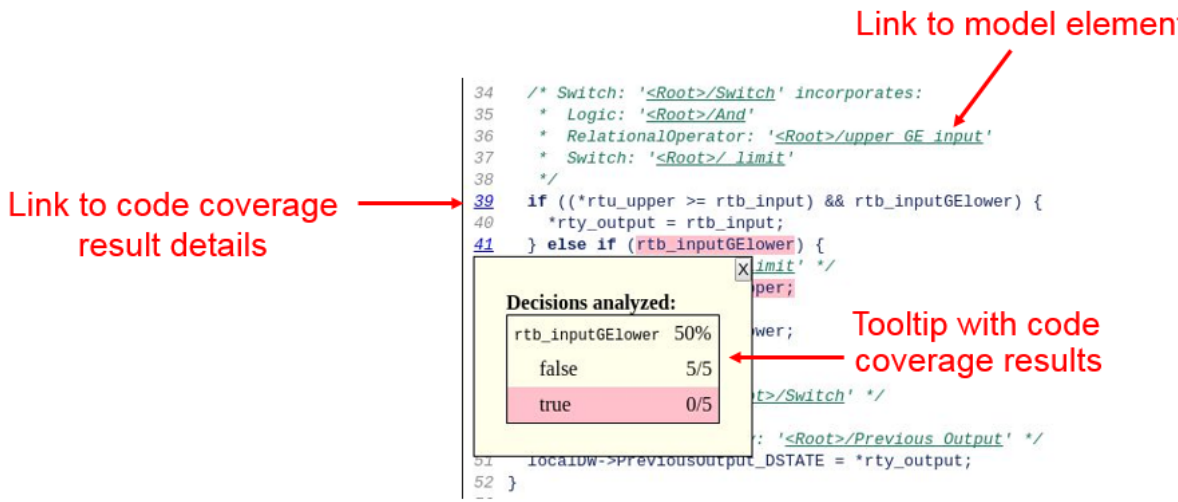
Reviewing the Coverage Results for Models in SIL or PIL Mode

In the code coverage report, each hyperlink opens a report with more details on the coverage analysis for the model. The code coverage results in these reports are similar to

the coverage results for C/C++ code in S-function blocks, as described in "View Coverage Results for C/C++ Code in S-Function Blocks" (Simulink Verification and Validation). You can navigate from code coverage results to the associated model blocks by using the links within the detailed code coverage reports.



Each detailed code coverage report also contains syntax highlighted code with coverage information.



Limitations

Coverage for models in SIL and PIL mode has these limitations:

- The model must meet the requirements listed in "Requirements to Enable SIL or PIL Code Coverage for a Model" on page 67-6.
- Code coverage results must not include external C/C++ files in read-only folders.

Related Examples

- "Software-in-the-Loop Code Coverage" (Simulink Verification and Validation)

Configure Code Coverage with Third-Party Tools

During a top-model or Model block SIL or PIL simulation, you can collect code coverage metrics for generated code using a third-party tool. Embedded Coder supports the following tools:

- LDRA Testbed from LDRA Software Technology. For information about installing and using this tool, go to www.ldra.com.

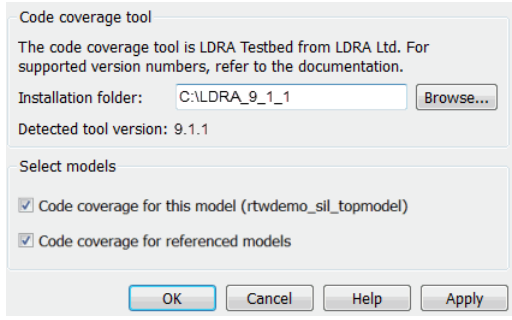
  The software supports LDRA Testbed code coverage for SIL and PIL.

- BullseyeCoverage from Bullseye Testing Technology. For information about installing and using this tool, go to www.bullseye.com.

  The software supports BullseyeCoverage code coverage for SIL and, in certain cases, PIL.

To configure a code coverage tool for a top-model or Model block SIL or PIL simulation:

1. Select **Simulation** > **Model Configuration Parameters** > **Code Generation** > **Verification**.
2. From the **Code coverage tool** drop-down list, select a tool, for example, `BullseyeCoverage` or `LDRA Testbed`.
3. Click **Configure Coverage** to open the Code Coverage Settings dialog box.
4. In the **Installation folder** field, specify the location where your coverage tool is installed. If you click **Browse**, the Select Installation Folder dialog box opens, which allows you to navigate to the folder where your coverage tool is installed. The software detects and displays the tool version.



By default, the software selects the following check boxes:

- **Code coverage for this model** — Generate coverage data for the current (top) model.
- **Code coverage for referenced models** — Generate data for models referenced by the current (top) model.

If your top model has Model blocks where the **Code interface** block parameter is set to `Top model`, then the top model and referenced models must have the same settings for these parameters. Otherwise, the software produces an error.

5 Click **OK**. You return to the **Verification** pane.

6 To view cumulative code coverage results within a code generation report, in the **Configuration Parameters** > **Code Generation** > **Report** pane, select the following check boxes:

- **Create code generation report**
- **Launch report automatically**

7 Click **OK**. You return to the model window.

With LDRA Testbed:

- The evaluation of cumulative code coverage begins from the point when you last added a new file to the existing set of source files. For example, existing code coverage results are deleted when you:
  - Run a simulation with a new model using the existing code generation folder.
  - Run a simulation that results in additional source code files being instrumented.
- If you switch between SIL and PIL simulations of a model, the software generates separate cumulative code coverage results for the SIL and PIL simulations.

For a model in a reference hierarchy, the software does not support simultaneous function execution time measurement and code coverage.

Related Examples

- “Configure and Run SIL Simulation” on page 64-15
- “Configure Code Coverage Programmatically” on page 67-16
- “View Code Coverage Information at the End of SIL or PIL Simulations” on page 67-13

- "Collect Code Coverage Metrics with a Third-Party Tool"
- "Code Coverage Tool Support" on page 67-23
- "Minor SIL and PIL Differences for LDRA Testbed" on page 67-26
- "PIL Support for BullseyeCoverage" on page 67-27
- "Simulink Code Coverage Metrics" on page 67-2
- "Code Coverage for Models in Software-in-the-Loop (SIL) Mode and Processor-in-the-Loop (PIL) Mode" on page 67-6

External Websites

- www.ldra.com
- www.bullseye.com

View Code Coverage Information at the End of SIL or PIL Simulations

In this section...

If you configure third-party code coverage for a SIL or PIL simulation, when the simulation is complete, the code generation report opens automatically and you see hyperlinks in the Command Window.

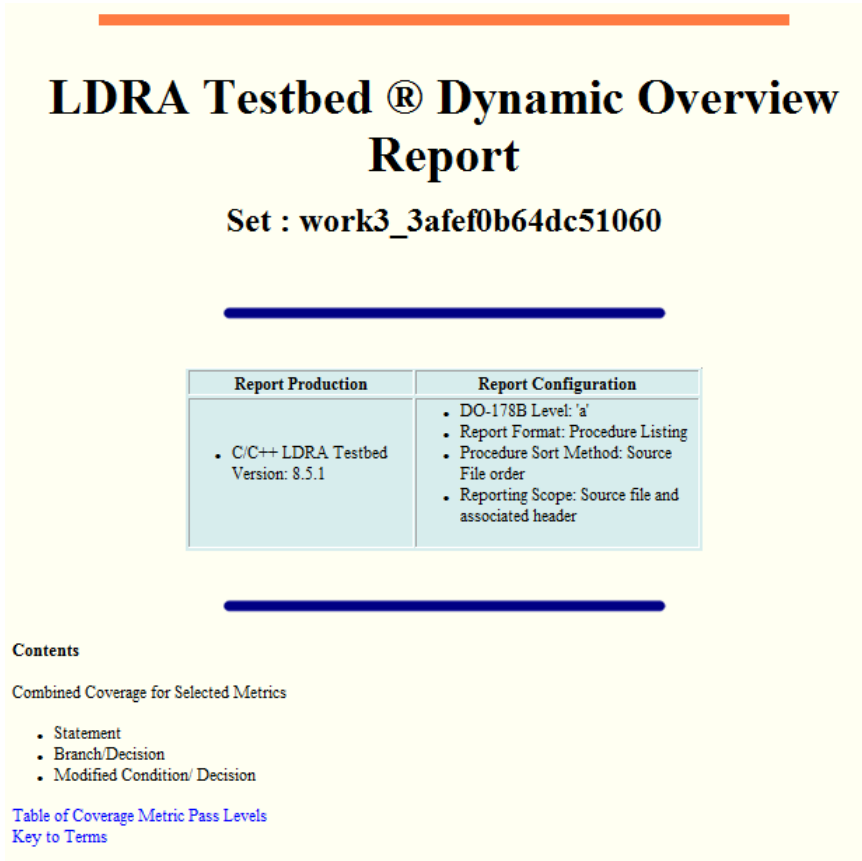
View LDRA Testbed Results

If you specified the LDRA Testbed, you see three links in the Command Window:

```
### Starting SIL simulation for component: rtwdemo_sil_topmodel
### Stopping SIL simulation for component: rtwdemo_sil_topmodel
### Starting analysis of coverage data
### Use the following links to view code coverage results:
    LDRA Testbed GUI
    LDRA Testbed Code Coverage Overview Report
    HTML code generation report with code coverage annotations
### Completed code coverage analysis
>>
```

To go to the LDRA Testbed GUI, click the `LDRA Testbed GUI` link.

To open the LDRA Testbed Report with your Web browser, click the `LDRA Testbed Code Coverage Overview Report` link.



LDRA Testbed ® Dynamic Overview Report

Set : work3_3afef0b64dc51060

Report Production	Report Configuration
• C/C++ LDRA Testbed Version: 8.5.1	• DO-178B Level: 'a' • Report Format: Procedure Listing • Procedure Sort Method: Source File order • Reporting Scope: Source file and associated header

Contents

Combined Coverage for Selected Metrics

- Statement
- Branch/Decision
- Modified Condition/ Decision

Table of Coverage Metric Pass Levels
Key to Terms

For information about using this report, refer to the LDRA Testbed documentation.

LDRA Testbed analysis results for all code in the current code generation folder belong to a set. In this set, you can find analysis results for models that share the same code generation folder. The `LDRA Testbed Code Coverage Overview Report` link identifies the location of the LDRA Testbed analysis results, which is determined by:

- The LDRA Testbed configuration.
- The name of the LDRA Testbed set associated with the current code generation folder.

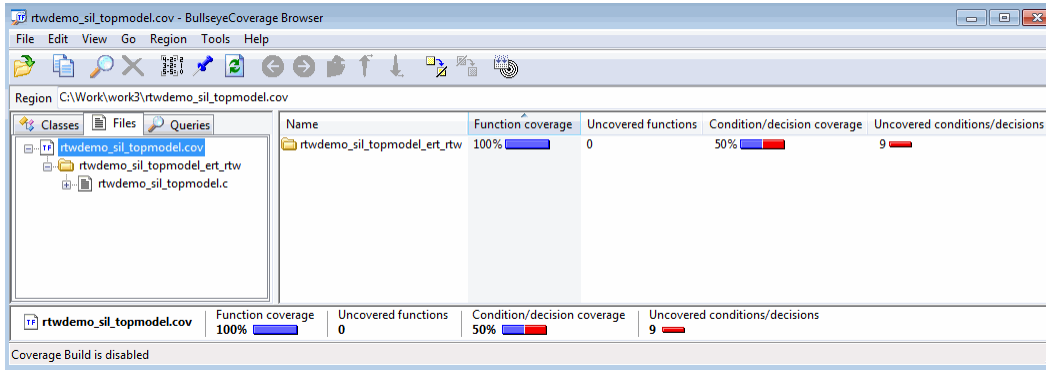
To view summary data and code annotations with coverage information in the code generation report, click the `HTML code generation report with code coverage annotations` link.

View BullseyeCoverage Results

If you specified the BullseyeCoverage tool, you see two links in the Command Window:

```
### Starting SIL simulation for component: rtwdemo_sil_topmodel
### Stopping SIL simulation for component: rtwdemo_sil_topmodel
### Processing code coverage data
### Use the following links to view code coverage results:
    BullseyeCoverage browser (coverage for last run)
    HTML code generation report (cumulative coverage)
### Completed code coverage analysis
>>
```

To view the coverage report using the BullseyeCoverage Browser, click the `BullseyeCoverage browser (coverage for last run)` link.



The BullseyeCoverage Browser shows coverage data for instrumented files associated with your latest top-model simulation. The coverage data shown in the browser is not cumulative and pertains only to the most recent simulation. For information about the BullseyeCoverage Browser, go to www.bullseye.com.

To view summary data and code annotations with coverage information in the code generation report, click the `HTML code generation report (cumulative coverage)` link.

Related Examples

- “Configure Code Coverage with Third-Party Tools” on page 67-10
- “Collect Code Coverage Metrics with a Third-Party Tool”
- “Code Coverage Summary and Annotations” on page 67-18

Configure Code Coverage Programmatically

You can configure code coverage for your model using command-line APIs. A typical workflow with BullseyeCoverage is:

1. Using `get_param`, retrieve the object containing coverage settings for the current model, for example, `gcs`.

   ```
   >> covSettings = get_param(gcs, 'CodeCoverageSettings')

   covSettings =

     CodeCoverageSettings with properties:

             TopModelCoverage: 'on'
       ReferencedModelCoverage: 'off'
                  CoverageTool: 'BullseyeCoverage'
   ```

   The property `TopModelCoverage` determines whether the software generates code coverage data for just the top model, while `ReferencedModelCoverage` determines whether the software generates coverage data for models referenced by the top model. If neither property is `'on'`, then no code coverage data is generated during a SIL simulation.

   If LDRA Testbed is the specified code coverage tool, then the property `CoverageTool` is `'LDRA Testbed'`.

   When you save your model, the properties `TopModelCoverage`, `ReferencedModelCoverage`, and `CoverageTool` are also saved.

2. Check the class of `covSettings`.

   ```
   >> class(covSettings)

   ans =

   coder.coverage.CodeCoverageSettings
   ```

3. Turn on coverage for referenced models.

   ```
   >> covSettings.ReferencedModelCoverage='on';
   ```

4. Using `set_param`, apply the new coverage settings to the model.

   ```
    >>set_param(gcs,'CodeCoverageSettings', covSettings);
   ```

5. Assuming you have installed the BullseyeCoverage tool, specify the installation path.

   ```
   >>coder.coverage.BullseyeCoverage.setPath('C:\Program Files\BullseyeCoverage')
   ```

For LDRA Testbed, use `coder.coverage.LDRA.setPath('C:\...)`.

6 Check that the path is saved as a preference.

```
>> coder.coverage.BullseyeCoverage.getPath
```

For LDRA Testbed, use `coder.coverage.LDRA.getPath`.

Related Examples

- "Collect Code Coverage Metrics with a Third-Party Tool"
- "Configure Code Coverage with Third-Party Tools" on page 67-10

Code Coverage Summary and Annotations

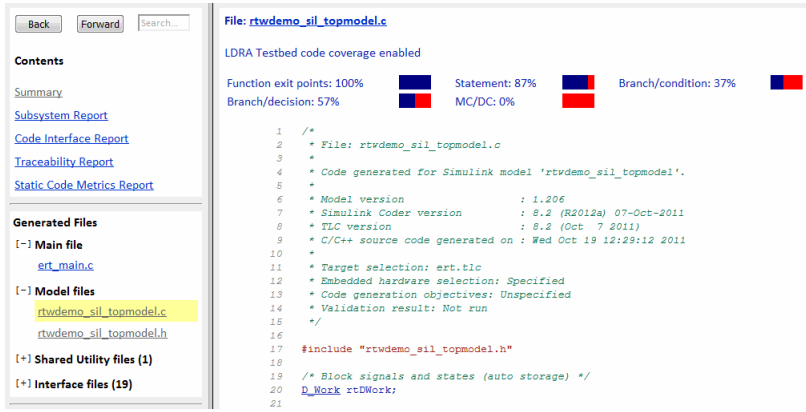
In this section...

If you specify a code coverage tool for a SIL or PIL simulation, the software produces a code generation report that provides summary data and code annotations with coverage information. Each code annotation is associated with a code feature and indicates the nature of the feature coverage during code execution.

The code generation report also allows you to navigate easily between blocks in your model and the corresponding sections in the source code.

LDRA Testbed Coverage

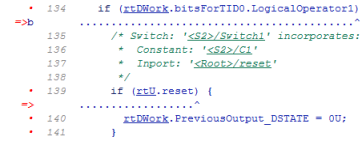
The cumulative coverage data in a code generation report is derived from instrumented files associated with your latest top-model simulation **and** coverage data collected from simulations with other top models that share referenced models with your current top model.



The software provides LDRA Testbed annotations in the code generation report to help you to review code coverage.

Note: Do not use the code generation report alone to verify that you have achieved your coverage goals. You must refer to the LDRA Testbed Report.

This example shows three kinds of annotations. On lines 134, 139, 140, and 141, the annotation • indicates that statement coverage for each of these lines of code is not complete.



Placing the cursor over the annotation =>b produces a tooltip.

This tooltip indicates that only one branch destination is covered. The code within the curly brackets, which starts at column 45 of line 134, is not executed. As the `if` statement on line 139 lies within this code, the corresponding annotation => states that the branch is not covered.

The following table describes the LDRA Testbed code annotations that you might see in a code generation report produced by a SIL and PIL simulations.

Code feature	Annotation symbol	What happened during simulation
Function	Fcn	Function *name* returned through this exit point.
	=>	Function *name* never returned through this exit point.
Branch/condition	=>	Condition not encountered.
	=>t	Condition evaluated true only.
	=>f	Condition evaluated false only.

Code feature	Annotation symbol	What happened during simulation
	tf	Condition evaluated both true and false.
Branch/decision	=>	Branch never encountered.
	=>b	Branch to at least one destination covered and branch to at least one other destination not covered.
	b	Branch fully exercised.
Modified Condition/ Decision Coverage (MC/DC)	=>mc	Condition did not independently affect outcome of decision.
	mc	Condition independently affected outcome of decision.
Statement		Statements associated with line covered.
		Not all statements associated with line covered.
Code that is reformatted by LDRA Testbed and does not match the original source code. For example, source code with `#include` statements to include other files, and source code with `#define` statements for macros. For detailed coverage information, refer to the LDRA Testbed report.	=>Σ	Zero coverage — probes within source code line or files included by source code line not exercised.
	=>Σ	Coverage probes within source code line or any included file partially exercised.
	Σ	Coverage probes within source code line or included files fully exercised.

BullseyeCoverage Information

The cumulative coverage data in a code generation report is derived from instrumented files associated with your latest top-model simulation **and** coverage data collected from

simulations with other top models that share referenced models with your current top model.

The software provides BullseyeCoverage annotations in the code generation report to help you to review code coverage.

This example shows two kinds of annotations. At line 41, TF indicates that the `if` decision had both true and false outcomes during the simulation. At line 52, =>F indicates that the `if` decision was false only during the simulation.

```
TF      41    if (rtU.reset) {
        42      rtDWork.PreviousOutput_DSTATE = 20U;
        43    }
        44
        45    /* Switch: '<Root>/Switch' incorporates:
        46     *  Constant: '<Root>/C1'
        47     *  Constant: '<Root>/C5'
        48     *  Inport: '<Root>/ ticks_to_count'
        49     *  RelationalOperator: '<Root>/upper GE input'
        50     *  Sum: '<Root>/Add'
        51     */
 =>F    52    if ((uint8_T)((uint32_T)rtU.ticks_to_count + (uint32
        53                  rtDWork.PreviousOutput_DSTATE) == 40)
```

The following table describes the BullseyeCoverage code annotations that you might see in a code generation report produced by a SIL simulation.

Code feature	Annotation symbol	What happened during simulation
Decision	=>	Decision not executed.
	TF	Decision evaluated both true and false.

Code feature	Annotation symbol	What happened during simulation
	=>T	Decision evaluated true only.
	=>F	Decision evaluated false only.
Function	=>	Function not called.
	Fcn	Function called.
Switch label	=>	Switch command not used.
	Sw	Switch command used.
Constant	k	Decision or condition was constant, which did not allow any variation in coverage.
Condition	=>	Condition not encountered.
	tf	Condition evaluated both true and false.
	=>t	Condition evaluated true only.
	=>f	Condition evaluated false only.
Try	=>	Try block never completed.
	Try	Try block covered.
Catch	=>	Catch block not covered.
	Cat	Catch block covered.

Related Examples

- “Configure Code Coverage with Third-Party Tools” on page 67-10
- “View Code Coverage Information at the End of SIL or PIL Simulations” on page 67-13
- “Trace Model Objects to Generated Code” on page 61-8
- “Trace Code to Model Objects by Using Hyperlinks” on page 61-6
- “Collect Code Coverage Metrics with a Third-Party Tool”

Code Coverage Tool Support

Embedded Coder code coverage provides the following support for the BullseyeCoverage and LDRA Testbed tools.

	BullseyeCoverage		LDRA Testbed	
Operating system	**Version supported**	**Compiler supported**	**Version supported**	**Compiler supported**
Windows	8.9.37	Microsoft Visual C++ (MSVC)	9.4.6	• Microsoft Visual C++ (MSVC) • LCC • MinGW
Linux	8.9.37	`gcc`	9.4.6	`gcc`
Mac	Not supported		Not supported	

Related Examples

- “Configure Code Coverage with Third-Party Tools” on page 67-10
- “Select and Configure C or C++ Compiler or IDE” (Simulink Coder)

Tips and Limitations

Right-Click Subsystem Build Unsupported for Code Coverage

The software does not support right-click builds for subsystems if a code coverage tool is specified.

BullseyeCoverage License Wait

When you build your model, you might have to wait for a BullseyeCoverage license. If you want to see information about the wait, before you build your model, select **Configuration Parameters > All Parameters > Verbose build**.

Current Working Folder Cannot be UNC Path

If your MATLAB current working folder is a Universal Naming Convention (UNC) path, code coverage fails.

Characters in `matlabroot` and File Path

If `matlabroot` or the path to your generated files contains a space or the `.` (period) character, code coverage might fail.

Header Files with Identical Names

Consider a model that is configured for LDRA Testbed code coverage. During the build process, if the software detects two header files with the same name in the folder for generated code, the software generates an error.

Code Coverage for Source Files in Shared Utility Folders

The software supports code coverage for source files generated in shared utility folders. If you configure code coverage for a model that uses shared utility code generation, when you build the model, you also build all source files in the shared utilities folder with code coverage enabled.

Whenever you build a model, the code coverage settings of the model must be consistent with source files that you previously built in the shared utilities folder. Otherwise, the

software reports that code in the shared utilities folder is inconsistent with the current model configuration and must be rebuilt. For example, if you run a SIL simulation for a model with code coverage enabled and then run a SIL simulation for another model with code coverage disabled, the software must rebuild all source files in the shared utilities folder.

BullseyeCoverage Behavior with Inline Macros

The BullseyeCoverage tool, by default, does not provide code coverage data for inline macros.

For example, if a model generates a file `slprj/ert/_sharedutils/rt_SATURATE.h` that contains the macro

```
#define rt_SATURATE(sig,ll,ul) (((sig) >= (ul)) ? (ul) : (((sig) <=
(ll)) ? (ll) : (sig)) )
```

and the macro is in `sat_ert_rtw/sat.c`, then the coverage report provides a measurement for `sat.c`, but no coverage data for the conditions within the macro `rt_SATURATE`.

To configure the BullseyeCoverage tool to provide code coverage data for inline macros:

1. Open the BullseyeCoverage Browser.
2. Select **Tools** > **Options** to open the Options dialog box.
3. On the **Build** tab, select the **Instrument macro expansions** check box.
4. Click **OK**.
5. Rerun your simulation.

Alternatively, you can add the text `-macro` to the `BullseyeCoverage` configuration file. For more information, go to www.bullseye.com/help.

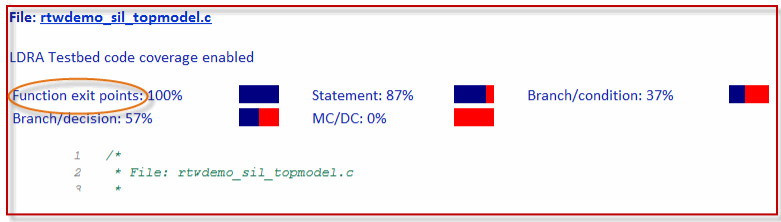
SIL and PIL Simulations with Open LDRA Testbed

If you enable code coverage with the LDRA Testbed tool, you must verify that the LDRA Testbed GUI is not open when you run your SIL or PIL simulation. If the set name in the LDRA Testbed GUI differs from the set name used by the SIL or PIL simulation, the SIL or PIL simulation fails.

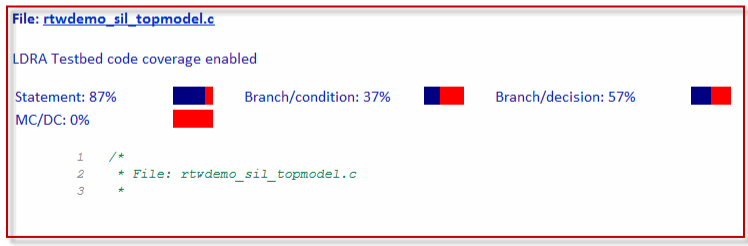
Minor SIL and PIL Differences for LDRA Testbed

The target connectivity API supports code coverage with LDRA Testbed for top-model and Model block PIL.

There are minor differences in the code coverage information collected during SIL and PIL simulations. In particular, with PIL, the software does not explicitly show function exit point coverage. However, you can infer the coverage of function exit points by examining statement coverage.



SIL



PIL

PIL Zero Coverage LDRA Testbed Annotations

For a PIL simulation with LDRA Testbed code coverage specified, there might be some source files where the recorded coverage is zero. In this case, the software provides summary information indicating that:

- There is coverage to measure.
- The coverage is zero.

You do not see information for individual probes on each line. The displayed summary information has an associated annotation tooltip:

```
0 out of N coverage probes were exercised (detailed breakdown unavailable)
```

PIL Support for BullseyeCoverage

Code coverage with BullseyeCoverage is available for top-model and Model block PIL provided your PIL application can write directly to the host file system. Your target for the PIL application must provide `fopen` and `fread` access to the host file system.

If code coverage is not available when you run the PIL application on your target hardware, you might be able to collect code coverage measurements by running the PIL application on an instruction set simulator that supports direct file I/O with the host file system.

Modify Legacy Code

If you modify legacy code and rerun a SIL or PIL simulation, the legacy code is recompiled. However, the code from the model may be up-to-date. In this case, the code generation report is not updated and does not show the modified legacy code. Instead, the code coverage information for the modified legacy code is displayed with reference to the original legacy code. You must regenerate the report. For more information, see "Limitation" (Simulink Coder).

IDE Link Does Not Support LDRA Testbed

When you generate code for IDE Link, you cannot use LDRA Testbed for SIL or PIL code coverage. Specifically, this limitation applies when you use the following settings together:

- **Configuration Parameters** > **Code Generation** > **System target file**: `idelink_ert.tlc`
- **Configuration Parameters** > **Code Generation** > **Verification** > **Code coverage tool**: `LDRA Testbed`.

Embedded IDEs and Embedded Targets

Getting Started with Embedded Targets in Embedded Coder

Embedded Coder Supported Hardware

As of this release, Embedded Coder supports the following hardware.

Support Package	Vendor	Earliest Release Available	Last Release Available
Altera SoC	Altera®	R2014b	Current
Analog Devices DSPs	Analog Devices®	R2013a	R2015b
ARM Cortex-A Processors	ARM®	R2014a	Current
ARM Cortex-M Processors	ARM	R2013b	Current
ARM Cortex-R Processors	ARM	R2016b	Current
AUTOSAR Standard	AUTOSAR (AUTomotive Open System ARchitecture) development partnership	R2014b	Current
BeagleBone Black Hardware	BeagleBoard	R2014b	Current
Green Hills MULTI	Green Hills® Software	R2012b	R2014a
STMicroelectronics Discovery Boards	STMicroelectronics®	R2013b	Current
Texas Instruments C2000 Processors	Texas Instruments	R2013b	Current
Texas Instruments C2000 F28M3x Concerto Processors	Texas Instruments	R2014b	Current
Texas Instruments C6000 DSPs	Texas Instruments	R2014a	R2016a
Wind River VxWorks RTOS	Wind River	R2013b	Current
Xilinx Zynq-7000 Platform	Xilinx®	R2013a	Current

For a complete list of supported hardware, see Hardware Support.

Run-Time Data Interface Extensions in Simulink Coder

Customize Generated ASAP2 File

In this section...

About ASAP2 File Customization

The Embedded Coder product provides a number of Target Language Compiler (TLC) files to enable you to customize the ASAP2 file generated from a Simulink model.

ASAP2 File Structure on the MATLAB Path

The ASAP2 related files are organized within the folders identified below:

- TLC files for generating ASAP2 file

  The *`matlabroot`*`/rtw/c/tlc/mw` (open) folder contains TLC files that generate ASAP2 files, `asamlib.tlc`, `asap2lib.tlc`, `asap2main.tlc`, and `asap2grouplib.tlc`. These files are included by the selected **System target file**. (See “Targets Supporting ASAP2” on page 44-3.)

- ASAP2 target files

  The *`matlabroot`*`/toolbox/rtw/targets/asap2/asap2` (open) folder contains the ASAP2 system target file and other control files.

- Customizable TLC files

  The *`matlabroot`*`/toolbox/rtw/targets/asap2/asap2/user` (open) folder contains files that you can modify to customize the content of your ASAP2 files.

- ASAP2 templates

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

Customize the Contents of the ASAP2 File

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

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

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

The procedure for customizing the ASAP2 file is as follows:

1. Make a copy of the asap2/user folder before making modifications.
2. Remove the old asap2/user folder from the MATLAB path, or add the new asap2/user folder to the MATLAB path above the old folder. The MATLAB session uses the ASAP2 setup file, asap2setup.tlc, in the new folder.

   asap2setup.tlc specifies the folders and files to include in the TLC path during the ASAP2 file generation process. Modify asap2setup.tlc to control the folders and folders included in the TLC path.
3. Modify the static parts of the ASAP2 file. These include

   - Project and header symbols, which are specified in asap2setup.tlc
   - Static sections of the file, such as file header and tail, A2ML, MOD_COMMON, and so on These are specified in asap2userlib.tlc.
   - Specify the appearance of the dynamic contents of the ASAP2 file by modifying the existing ASAP2 templates or by defining new ASAP2 templates. Sections of the ASAP2 file affected include

     RECORD_LAYOUT: modify parts of the ASAP2 template files.

`CHARACTERISTIC`: modify parts of the ASAP2 template files. For more information on modifying the appearance of `CHARACTERISTIC` records, see "ASAP2 Templates" on page 69-4.

- `MEASUREMENT`: These are specified in `asap2userlib.tlc`.
- `COMPU_METHOD`: These are specified in `asap2userlib.tlc`.

ASAP2 Templates

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

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

1. Create a template definition file. See "Create Template Definition Files" on page 69-4.
2. Include the template definition file in the TLC path. The path is specified in the ASAP2 setup file, `asap2setup.tlc`.

Create Template Definition Files

This section describes the components that make up an ASAP2 template definition file. This description is in the form of code examples from `asap2lookup1d.tlc`, the template definition file for the `Lookup1D` template. This template corresponds to the `Lookup1D` parameter group.

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

Template Registration Function

The input argument is the name of the parameter group associated with this template:

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

RECORD_LAYOUT Name Definition Function

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

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

Function to Write RECORD_LAYOUT Definitions

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

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

Function to Write the CHARACTERISTIC

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

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

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

```
%function ASAP2UserFcnWriteCharacteristic_Lookup1D(paramGroup)
Output
```

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

Customize Computation Method Names

In generated ASAP2 files, computation methods translate the electronic control unit (ECU) internal representation of measurement and calibration quantities into a physical model oriented representation. Simulink Coder software provides the ability to customize the names of computation methods. You can provide names that are more intuitive, enhancing ASAP2 file readability, or names that meet organizational requirements.

To customize computation method names, use the MATLAB function `getCompuMethodName`, which is defined in *`matlabroot`*`/toolbox/rtw/targets/asap2/asap2/user/getCompuMethodName.m`.

The `getCompuMethodName` function constructs a computation method name. The function prototype is

```
cmName = getCompuMethodName(dataTypeName, cmUnits)
```

where *`dataTypeName`* is the name of the data type associated with the computation method, *`cmUnits`* is the units as specified in the `Unit` property of a `Simulink.Parameter` or `Simulink.Signal` object (for example, `rpm` or `m/s`), and *`cmName`* returns the constructed computation method name.

The default constructed name returned by the function has the format

```
<localPrefix><datatype>_<cmUnits>
```

where

- `<local_Prefix>` is a local prefix, `CM_`, defined in *`matlabroot`*`/toolbox/rtw/targets/asap2/asap2/user/getCompuMethodName.m`.
- `<datatype>` and `<cmUnits>` are the arguments you specified to the `getCompuMethodName` function.

Additionally, in the generated ASAP2 file, the constructed name is prefixed with <ASAP2CompuMethodName_Prefix>, a model prefix defined in *matlabroot*/toolbox/rtw/targets/asap2/asap2/user/asap2setup.tlc.

For example, if you call the getCompuMethodName function with the *dataTypeName* argument 'int16' and the *cmUnits* argument 'm/s', and generate an ASAP2 file for a model named myModel, the computation method name would appear in the generated file as follows:

```
/begin COMPU_METHOD
   /* Name of CompuMethod */ myModel_CM_int16_m_s
   /* Units */ "m/s"
   ...
/end COMPU_METHOD
```

Suppress Computation Methods for FIX_AXIS

Versions 1.51 and later of the ASAP2 specification state that for certain cases of lookup table axis descriptions (integer data type and no doc units), a computation method is not required and the Conversion Method parameter must be set to the value NO_COMPU_METHOD. You can control whether or not computation methods are suppressed when not required using the Target Language Compiler (TLC) option ASAP2GenNoCompuMethod. This TLC option is disabled by default. If you enable the option, ASAP2 file generation does not generate computation methods for lookup table axis descriptions when not required, and instead generates the value NO_COMPU_METHOD. For example:

```
/begin CHARACTERISTIC
/* Name            */
lu1d_fix_axisTable_data
...
/begin AXIS_DESCR
    ...
    /* Conversion Method    */
NO_COMPU_METHOD
    ...
/end CHARACTERISTIC
```

The ASAP2GenNoCompuMethod option is defined in *matlabroot*/toolbox/rtw/targets/asap2/asap2/user/asap2setup.tlc.

Create a Transport Layer for External Communication

In this section...

About Creating a Transport Layer for External Communication

This section helps you to connect your custom target by using external mode using your own low-level communications layer. The topics include:

- An overview of the design and operation of external mode
- A description of external mode source files
- Guidelines for modifying the external mode source files and building an executable to handle the tasks of the default `ext_comm` MEX-file

This section assumes that you are familiar with the execution of Simulink Coder programs, and with the basic operation of external mode.

Design of External Mode

External mode communication between the Simulink engine and a target system is based on a client/server architecture. The client (the Simulink engine) transmits messages requesting the server (target) to accept parameter changes or to upload signal data. The server responds by executing the request.

A low-level *transport layer* handles physical transmission of messages. Both the Simulink engine and the model code are independent of this layer. Both the transport layer and code directly interfacing to the transport layer are isolated in separate modules that format, transmit, and receive messages and data packets.

This design makes it possible for different targets to use different transport layers. The GRT, ERT, and RSim targets support host/target communication by using TCP/IP and

RS-232 (serial) communication. The Simulink Desktop Real-Time target supports shared memory communication. The Wind River Systems Tornado® target supports TCP/IP only.

The Simulink Coder product provides full source code for both the client and server-side external mode modules, as used by the GRT, ERT, Rapid Simulation, and Tornado targets, and the Simulink Desktop Real-Time and Simulink Real-Time products. The main client-side module is `ext_comm.c`. The main server-side module is `ext_svr.c`.

These two modules call the specified transport layer through the following source files.

Built-In Transport Layer Implementations

Protocol	Client or Server?	Source Files
TCP/IP	Client (host)	• *matlabroot*`/toolbox/coder/simulinkcoder_core/ext_mode/host/common/rtiostream_interface.c` • *matlabroot*`/rtw/c/src/rtiostream/rtiostreamtcpip/rtiostream_tcpip.c`
	Server (target)	• *matlabroot*`/rtw/c/src/ext_mode/common/rtiostream_interface.c` • *matlabroot*`/rtw/c/src/rtiostream/rtiostreamtcpip/rtiostream_tcpip.c`
Serial	Client (host)	• *matlabroot*`/toolbox/coder/simulinkcoder_core/ext_mode/host/serial/ext_serial_transport.c` • *matlabroot*`/rtw/c/src/rtiostream/rtiostreamserial/rtiostream_serial.c`
	Server (target)	• *matlabroot*`/rtw/c/src/ext_mode/serial/ext_svr_serial_transport.c` • *matlabroot*`/rtw/c/src/rtiostream/rtiostreamserial/rtiostream_serial.c`

For serial communication, the modules `ext_serial_transport.c` and `rtiostream_serial.c` implement the client-side transport functions and the modules `ext_svr_serial_transport.c` and `rtiostream_serial.c` implement the corresponding server-side functions. For TCP/IP communication, the modules `rtiostream_interface.c` and `rtiostream_tcpip.c` implement both client-side and server-side functions. You can edit copies of these files (but do not modify the originals).

You can support external mode using your own low-level communications layer by creating similar files using the following templates:

- Client (host) side: *matlabroot*/rtw/c/src/rtiostream/rtiostreamtcpip/rtiostream_tcpip.c (TCP/IP) or *matlabroot*/rtw/c/src/rtiostream/rtiostreamserial/rtiostream_serial.c (serial)
- Server (target) side: *matlabroot*/rtw/c/src/rtiostream/rtiostreamtcpip/rtiostream_tcpip.c (TCP/IP) or *matlabroot*/rtw/c/src/rtiostream/rtiostreamserial/rtiostream_serial.c (serial)

The file `rtiostream_interface.c` is an interface between the external mode protocol and an `rtiostream` communications channel. For more details on implementing an `rtiostream` communications channel, see "Communications `rtiostream` API" on page 64-46. Implement your `rtiostream` communications channel by using the documented interface to avoid having to make changes to the file `rtiostream_interface.c` or other external mode related files.

Note Do not modify working source files. Use the templates provided in the `/custom` or `/rtiostream` folder as starting points, guided by the comments within them.

You need only provide code that implements low-level communications. You need not be concerned with issues such as data conversions between host and target, or with the formatting of messages. The Simulink Coder software handles these functions.

On the client (Simulink engine) side, communications are handled by `ext_comm` (for TCP/IP) and `ext_serial_win32_comm` (for serial) MEX-files.

On the server (target) side, external mode modules are linked into the target executable. This takes place automatically if the **External mode** code generation option is selected at code generation time, based on the **External mode transport** option selected in the target code generation options dialog box. These modules, called from the main program and the model execution engine, are independent of the generated model code.

The general procedure for implementing your own client-side low-level transport protocol is as follows:

1. Edit the template `rtiostream_tcpip.c` to replace low-level communication calls with your own communication calls.
2. Generate a MEX-file executable for your custom transport.

3 Register your new transport layer with the Simulink software, so that the transport can be selected for a model using the **Interface** pane of the Configuration Parameters dialog box.

For more details, see "Create a Custom Client (Host) Transport Protocol" on page 69-17.

The general procedure for implementing your own server-side low-level transport protocol is as follows:

1 Edit the template `rtiostream_tcpip.c` to replace low-level communication calls with your own communication calls. Typically this involves writing or integrating device drivers for your target hardware.

2 Modify template makefiles to support the new transport.

For more details, see "Create a Custom Server (Target) Transport Protocol" on page 69-21.

External Mode Communications Overview

This section gives a high-level overview of how a Simulink Coder generated program communicates with Simulink external mode. This description is based on the TCP/IP version of external mode that ships with the Simulink Coder product.

For communication to take place,

- The server (target) program must have been built with the conditional `EXT_MODE` defined. `EXT_MODE` is defined in the *`model`*`.mk` file if the **External mode** code generation option was selected at code generation time.
- Both the server program and the Simulink software must be executing. This does not mean that the model code in the server system must be executing. The server can be waiting for the Simulink engine to issue a command to start model execution.

The client and server communicate by using bidirectional sockets carrying packets. Packets consist either of *messages* (commands, parameter downloads, and responses) or *data* (signal uploads).

If the target program was invoked with the `-w` command-line option, the program enters a wait state until it receives a message from the host. Otherwise, the program begins execution of the model. While the target program is in a wait state, the Simulink engine can download parameters to the target and configure data uploading.

When the user chooses the **Connect to Target** option from the **Simulation** menu, the host initiates a handshake by sending an `EXT_CONNECT message`. The server responds with information about itself. This information includes

- Checksums. The host uses model checksums to determine that the target code is an exact representation of the current Simulink model.
- Data format information. The host uses this information when formatting data to be downloaded, or interpreting data that has been uploaded.

At this point, host and server are connected. The server is either executing the model or in the wait state. (In the latter case, the user can begin model execution by selecting **Start Real-Time Code** from the **Simulation** menu.)

During model execution, the message server runs as a background task. This task receives and processes messages such as parameter downloads.

Data uploading comprises both foreground execution and background servicing of the signal packets. As the target computes model outputs, it also copies signal values into data upload buffers. This occurs as part of the task associated with each task identifier (`tid`). Therefore, data collection occurs in the foreground. Transmission of the collected data, however, occurs as a background task. The background task sends the data in the collection buffers to the Simulink engine by using data packets.

The host initiates most exchanges as messages. The target usually sends a response confirming that it has received and processed the message. Examples of messages and commands are:

- Connection message / connection response
- Start target simulation / start response
- Parameter download / parameter download response
- Arm trigger for data uploading / arm trigger response
- Terminate target simulation / target shutdown response

Model execution terminates when the model reaches its final time, when the host sends a terminate command, or when a Stop Simulation block terminates execution. On termination, the server informs the host that model execution has stopped, and shuts down its socket. The host also shuts down its socket, and exits external mode.

External Mode Source Files

- “Client (Host) MEX-file Interface Source Files” on page 69-13

- "Server (Target) Source Files" on page 69-14
- "Other Files in the Server Folder" on page 69-16

Client (Host) MEX-file Interface Source Files

The source files for the MEX-file interface component are located in the folder *matlabroot*`/toolbox/coder/simulinkcoder_core/ext_mode/host` (open), except as noted:

- `common/ext_comm.c`

  This file is the core of external mode communication. It acts as a relay station between the target and the Simulink engine. `ext_comm.c` communicates to the Simulink engine by using a shared data structure, `ExternalSim`. It communicates to the target by using calls to the transport layer.

  Tasks carried out by `ext_comm.c` include establishment of a connection with the target, downloading of parameters, and termination of the connection with the target.

- `common/rtiostream_interface.c`

  This file is an interface between the external mode protocol and an `rtiostream` communications channel. For more details on implementing an `rtiostream` communications channel, see "Communications `rtiostream` API" on page 64-46. Implement your `rtiostream` communications channel using the documented interface to avoid having to change the file `rtiostream_interface.c` or other external mode related files.

- *matlabroot*`/rtw/c/src/rtiostream/rtiostreamtcpip/rtiostream_tcpip.c`

  This file implements required TCP/IP transport layer functions. The version of `rtiostream_tcpip.c` shipped with the Simulink Coder software uses TCP/IP functions including `recv()`, `send()`, and `socket()`.

- *matlabroot*`/rtw/c/src/rtiostream/rtiostreamserial/rtiostream_serial.c`

  This file implements required serial transport layer functions. The version of `rtiostream_serial.c` shipped with the Simulink Coder software uses serial functions including `ReadFile()`, `WriteFile()`, and `CreateFile()`.

- `serial/ext_serial_transport.c`

This file implements required serial transport layer functions. `ext_serial_transport.c` includes `ext_serial_utils.c`, which is located in *`matlabroot`*`/rtw/c/src/ext_mode/serial` (open) and contains functions common to client and server sides.

- `common/ext_main.c`

  This file is a MEX-file wrapper for external mode. `ext_main.c` interfaces to the Simulink engine by using the standard `mexFunction` call. (See the `mexFunction` reference page and "MATLAB API for Other Languages" (MATLAB) for more information.) `ext_main.c` contains a function dispatcher, `esGetAction`, that sends requests from the Simulink engine to `ext_comm.c`.

- `common/ext_convert.c` and `ext_convert.h`

  This file contains functions used for converting data from host to target formats (and vice versa). Functions include byte-swapping (big to little- endian), conversion from non-IEEE floats to IEEE doubles, and other conversions. These functions are called both by `ext_comm.c` and directly by the Simulink engine (by using function pointers).

  Note You do not need to customize `ext_convert` to implement a custom transport layer. However, you might want to customize `ext_convert` for the intended target. For example, if the target represents the `float` data type in Texas Instruments format, `ext_convert` must be modified to perform a Texas Instruments to IEEE conversion.

- `common/extsim.h`

  This file defines the `ExternalSim` data structure and access macros. This structure is used for communication between the Simulink engine and `ext_comm.c`.

- `common/extutil.h`

  This file contains only conditionals for compilation of the `assert` macro.

- `common/ext_transport.h`

  This file defines functions that must be implemented by the transport layer.

Server (Target) Source Files

These files are linked into the *`model`*`.exe` executable. They are located within *`matlabroot`*`/rtw/c/src/ext_mode` (open) except as noted.

- `common/ext_svr.c`

  `ext_svr.c` is analogous to `ext_comm.c` on the host, but generally is responsible for more tasks. It acts as a relay station between the host and the generated code. Like `ext_comm.c`, `ext_svr.c` carries out tasks such as establishing and terminating connection with the host. `ext_svr.c` also contains the background task functions that either write downloaded parameters to the target model, or extract data from the target data buffers and send it back to the host.

- `common/rtiostream_interface.c`

  This file is an interface between the external mode protocol and an `rtiostream` communications channel. For more details on implementing an `rtiostream` communications channel, see "Communications `rtiostream` API" on page 64-46. Implement your `rtiostream` communications channel by using the documented interface to avoid having to change the file `rtiostream_interface.c` or other external mode related files.

- *`matlabroot`*`/rtw/c/src/rtiostream/rtiostreamtcpip/rtiostream_tcpip.c`

  This file implements required TCP/IP transport layer functions. The version of `rtiostream_tcpip.c` shipped with the Simulink Coder software uses TCP/IP functions including `recv()`, `send()`, and `socket()`.

- *`matlabroot`*`/rtw/c/src/rtiostream/rtiostreamserial/rtiostream_serial.c`

  This file implements required serial transport layer functions. The version of rtiostream_serial.c shipped with the software uses serial functions including `ReadFile()`, `WriteFile()`, and `CreateFile()`.

- *`matlabroot`*`/rtw/c/src/rtiostream.h`

  This file defines the `rtIOStream*` functions implemented in `rtiostream_tcpip.c`.

- `serial/ext_svr_serial_transport.c`

  This file implements required serial transport layer functions. `ext_svr_serial_transport.c` includes `serial/ext_serial_utils.c`, which contains functions common to client and server sides.

- `common/updown.c`

  `updown.c` handles the details of interacting with the target model. During parameter downloads, `updown.c` does the work of installing the new parameters into the

model's parameter vector. For data uploading, `updown.c` contains the functions that extract data from the model's `blockio` vector and write the data to the upload buffers. `updown.c` provides services both to `ext_svr.c` and to the model code (for example, `grt_main.c`). It contains code that is called by using the background tasks of `ext_svr.c` as well as code that is called as part of the higher priority model execution.

- *`matlabroot`*`/rtw/c/src/dt_info.h` (included by generated model build file *`model`*`.h`)

  These files contain data type transition information that allows access to multi-data type structures across different computer architectures. This information is used in data conversions between host and target formats.

- `common/updown_util.h`

  This file contains only conditionals for compilation of the `assert` macro.

- `common/ext_svr_transport.h`

  This file defines the `Ext*` functions that must be implemented by the server (target) transport layer.

Other Files in the Server Folder

- `common/ext_share.h`

  Contains message code definitions and other definitions required by both the host and target modules.

- `serial/ext_serial_utils.c`

  Contains functions and data structures for communication, MEX link, and generated code required by both the host and target modules of the transport layer for serial protocols.

- The serial transport implementation includes the additional files
  - `serial/ext_serial_pkt.c` and `ext_serial_pkt.h`
  - `serial/ext_serial_port.h`

Implement a Custom Transport Layer

- “Requirements for Custom Transport Layers” on page 69-17

Requirements for Custom Transport Layers

- By default, `ext_svr.c` and `updown.c` use `malloc` to allocate buffers in target memory for messages, data collection, and other purposes, although there is also an option to preallocate static memory. If your target uses another memory allocation scheme, you must modify these modules.
- The target is assumed to support both `int32_T` and `uint32_T` data types.

Create a Custom Client (Host) Transport Protocol

To implement the client (host) side of your low-level transport protocol,

1. Edit the template file *`matlabroot`*`/rtw/c/src/rtiostream/rtiostreamtcpip/rtiostream_tcpip.c` to replace low-level communication calls with your own communication calls.

   a. Copy and rename the file to `rtiostream_`*`name`*`.c` (replacing *`name`* with a name meaningful to you).

   b. Replace the functions `rtIOStreamOpen`, `rtIOStreamClose`, `rtIOStreamSend`, and `rtIOStreamRecv` with functions (of the same name) that call your low-level communication primitives. These functions are called from other external mode modules via `rtiostream_interface.c`. For more information, see "Communications `rtiostream` API" on page 64-46.

   c. Build your `rtiostream` implementation into a shared library that exports the `rtIOStreamOpen`, `rtIOStreamClose`, `rtIOStreamRecv` and `rtIOStreamSend` functions.

2. Build the customized MEX-file executable using the MATLAB `mex` function. See "MATLAB Commands to Rebuild ext_comm and ext_serial_win32 MEX-Files" on page 69-18 for examples of `mex` invocations.

   Do not replace the existing `ext_comm` MEX-file if you want to preserve its existing function. Instead, use the `-output` option to name the resulting

executable (for example, `mex -output ext_myrtiostream_comm ...` builds `ext_myrtiostream_comm.`*`mexext`*, on Windows platforms).

3 Register your new client transport layer with the Simulink software, so that the transport can be selected for a model using the **Interface** pane of the Configuration Parameters dialog box. For details, see "Register a Custom Client (Host) Transport Protocol" on page 69-20.

Sample commands for rebuilding external mode MEX-files are listed in "MATLAB Commands to Rebuild ext_comm and ext_serial_win32 MEX-Files" on page 69-18.

MATLAB Commands to Rebuild ext_comm and ext_serial_win32 MEX-Files

The following table lists the commands for building the standard `ext_comm` and `ext_serial_win32` modules on PC and UNIX platforms.

Platform	Commands

Windows, TCP/IP	`>> cd (matlabroot)`
	`>> mex toolbox\coder\simulinkcoder_core\ext_mode\host\common\ext_comm.c ...`
	`toolbox\coder\simulinkcoder_core\ext_mode\host\common\ext_convert.c ...`
	`toolbox\coder\simulinkcoder_core\ext_mode\host\common\rtiostream_interface.c ...`
	`toolbox\coder\simulinkcoder_core\ext_mode\host\common\ext_util.c ...`
	`-Irtw\c\src -Irtw\c\src\rtiostream\utils ...`
	`-Irtw\c\src\ext_mode\common ...`
	`-Itoolbox\coder\simulinkcoder_core\ext_mode\host\common ...`
	`-Itoolbox\coder\simulinkcoder_core\ext_mode\host\common\include ...`
	`-lmwrtiostreamutils -lsl_services ...`
	`-DEXTMODE_TCPIP_TRANSPORT ...`
	`-DSL_EXT_DLL -output toolbox\coder\simulinkcoder_core\ext_comm`
	
	Note: The `rtiostream_interface.c` function defines RTIOSTREAM_SHARED_LIB as `libmwrtiostreamtcpip` and dynamically loads the MathWorks TCP/IP `rtiostream` shared library. Modify this file if you need to load a different `rtiostream` shared library.

Linux, TCP/IP	Use the Windows commands, with these changes: • Change `-DSL_EXT_DLL` to `-DSL_EXT_SO`. • Change `-lsl_services` to `-lmwsl_services`. • Replace back slashes with forward slashes.
Mac, TCP/IP	Use the Windows commands, with these changes: • Change `-DSL_EXT_DLL` to `-DSL_EXT_DYLIB`.

Platform	Commands
	• Change `-lsl_services` to `-lmwsl_services`. • Replace back slashes with forward slashes.

Windows, serial	`>> cd (matlabroot)`
	`>> mex toolbox\coder\simulinkcoder_core\ext_mode\host\common\ext_comm.c ...`
	`toolbox\coder\simulinkcoder_core\ext_mode\host\common\ext_convert.c ...`
	`toolbox\coder\simulinkcoder_core\ext_mode\host\serial\ext_serial_transport.c ...`
	`toolbox\coder\simulinkcoder_core\ext_mode\host\serial\ext_serial_pkt.c ...`
	`toolbox\coder\simulinkcoder_core\ext_mode\host\serial\rtiostream_serial_interface.c ...`
	`toolbox\coder\simulinkcoder_core\ext_mode\host\common\ext_util.c ...`
	`-Irtw\c\src -Irtw\c\src\rtiostream\utils ...`
	`-Irtw\c\src\ext_mode\common ...`
	`-Irtw\c\src\ext_mode\serial ...`
	`-Itoolbox\coder\simulinkcoder_core\ext_mode\host\common ...`
	`-Itoolbox\coder\simulinkcoder_core\ext_mode\host\common\include ...`
	`-lmwrtiostreamutils -lsl_services ...`
	`-DEXTMODE_SERIAL_TRANSPORT -DSL_EXT_DLL ...`
	`-output toolbox\coder\simulinkcoder_core\ext_serial_win32_comm`
	
	Note: The `rtiostream_interface.c` function defines RTIOSTREAM_SHARED_LIB as `libmwrtiostreamserial` and dynamically loads the MathWorks serial `rtiostream` shared library. Modify this file if you need to load a different `rtiostream` shared library.

Linux, serial	Use the Windows commands, with these changes: • Change `-DSL_EXT_DLL` to `-DSL_EXT_SO`. • Change `-lsl_services` to `-lmwsl_services`. • Replace back slashes with forward slashes.
Mac, serial	Use the Windows commands, with these changes: • Change `-DSL_EXT_DLL` to `-DSL_EXT_DYLIB`. • Change `-lsl_services` to `-lmwsl_services`. • Replace back slashes with forward slashes.

Note: `mex` requires a compiler supported by the MATLAB API. See the `mex` reference page and "MATLAB API for Other Languages" (MATLAB) for more information about the `mex` function.

Register a Custom Client (Host) Transport Protocol

To register a custom client transport protocol with the Simulink software, you must add an entry of the following form to an `sl_customization.m` file on the MATLAB path:

```
function sl_customization(cm)
  cm.ExtModeTransports.add('stf.tlc', 'transport', 'mexfile', 'Level1');
% -- end of sl_customization
```

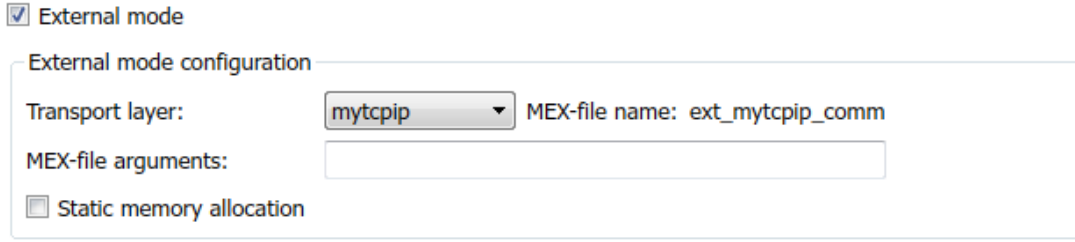
where

- *`stf`*`.tlc` is the name of the system target file for which the transport will be registered (for example, `'grt.tlc'`)
- *`transport`* is the transport name to display in the **Transport layer** menu on the **Interface** pane of the Configuration Parameters dialog box (for example, `'mytcpip'`)
- *`mexfile`* is the name of the transport's associated external interface MEX-file (for example, `'ext_mytcpip_comm'`)

You can specify multiple targets and/or transports with additional `cm.ExtModeTransports.add` lines, for example:

```
function sl_customization(cm)
  cm.ExtModeTransports.add('grt.tlc', 'mytcpip', 'ext_mytcpip_comm', 'Level1');
  cm.ExtModeTransports.add('ert.tlc', 'mytcpip', 'ext_mytcpip_comm', 'Level1');
% -- end of sl_customization
```

If you place the `sl_customization.m` file containing the transport registration information on the MATLAB path, your custom client transport protocol will be registered with each subsequent Simulink session. The name of the transport will appear in the **Transport layer** menu on the **Interface** pane of the Configuration Parameters dialog box. When you select the transport for your model, the name of the associated external interface MEX-file will appear in the noneditable **MEX-file name** field, as shown in the following figure.



Create a Custom Server (Target) Transport Protocol

The `rtIOStream*` function prototypes in *`matlabroot`*`/rtw/c/src/rtiostream.h` define the calling interface for both the server (target) and client (host) side transport layer functions.

- The TCP/IP implementations are in *`matlabroot`*`/rtw/c/src/rtiostream/rtiostreamtcpip/rtiostream_tcpip.c`.
- The serial implementations are in *`matlabroot`*`/rtw/c/src/rtiostream/rtiostreamserial/rtiostream_serial.c`.

Note: The `Ext*` function prototypes in *`matlabroot`*`/rtw/c/src/ext_mode/common/ext_svr_transport.h` are implemented in *`matlabroot`*`/rtw/c/src/ext_mode/common/rtiostream_interface.c` or *`matlabroot`*`/rtw/c/src/ext_mode/serial/rtiostream_serial_interface.c`. In most cases you will not need to modify `rtiostream_interface.c` or `rtiostream_serial_interface.c` for your custom TCP/IP or serial transport layer.

To implement the server (target) side of your low-level TCP/IP or serial transport protocol:

1. Edit the template *`matlabroot`*`/rtw/c/src/rtiostream/rtiostreamtcpip/rtiostream_tcpip.c` or *`matlabroot`*`/rtw/c/src/rtiostream/rtiostreamserial/rtiostream_serial.c` to replace low-level communication calls with your own communication calls.

   a. Copy and rename the file to `rtiostream_`*`name`*`.c` (replacing *`name`* with a name meaningful to you).

   b. Replace the functions `rtIOStreamOpen`, `rtIOStreamClose`, `rtIOStreamSend`, and `rtIOStreamRecv` with functions (of the same name) that call your low-level communication drivers.

      You must implement the functions defined in `rtiostream.h`, and your implementations must conform to the prototypes defined in that file. Refer to the original `rtiostream_tcpip.c` or `rtiostream_serial.c` for guidance.

2. Incorporate the external mode source files for your transport layer into the model build process, according to your target type:

- If your target uses toolchain controls to configure a build, use a build process mechanism such as a post code generation command or a `before_make` hook function to make the transport files available to the build process. (For more information on the build process mechanisms, see "Customize Post-Code-Generation Build Processing" (Simulink Coder), "Customize Build Process with STF_make_rtw_hook File" (Simulink Coder), and "Customize Build Process with sl_customization.m" (Simulink Coder).) For example:

  - Add the file created in the previous step to the build information:

    ```
    path/rtiostream_name.c
    ```

  - For TCP/IP, add the following file to the build information:

    ```
    matlabroot/rtw/c/src/ext_mode/common/rtiostream_interface.c
    ```

  - For serial, add the following files to the build information:

    ```
    matlabroot/rtw/c/src/ext_mode/serial/ext_serial_pkt.c
    matlabroot/rtw/c/src/ext_mode/serial/rtiostream_serial_interface.c
    matlabroot/rtw/c/src/ext_mode/serial/ext_svr_serial_transport.c
    ```

- If your target uses template makefile controls to configure a build, modify template makefiles to support the new transport. Be sure to include the file created in the previous step, `rtiostream_name.c`. If you are writing your own template makefile, make sure that the `EXT_MODE` code generation option is defined. The generated makefile will then link `rtiostream_name.c`, `rtiostream_interface.c` or `rtiostream_serial_interface.c`, and other server code into your executable.

Note: For external mode, check that `rtIOStreamRecv` is not a blocking implementation. Otherwise, it might cause the external mode server to block until the host sends data through the `comm` layer.

Serial Receive Buffer Smaller than 64 Bytes

For serial communication, if the serial receive buffer of your target is smaller than 64 bytes:

1 Update the following macro with the actual target buffer size:

```
#define TARGET_SERIAL_RECEIVE_BUFFER_SIZE 64
```

Implement the change in the following files:

```
matlabroot/rtw/c/src/ext_mode/serial/ext_serial_utils.c
matlabroot/toolbox/coder/simulinkcoder_core/ext_mode/host/serial/ext_serial_utils.c
```

2. Run the command to rebuild the `ext_serial_win32` MEX-file. See "MATLAB Commands to Rebuild ext_comm and ext_serial_win32 MEX-Files" on page 69-18.

Build Process Integration in Simulink Coder

Control Build Process Compiling and Linking

After generating code for a model, the build process determines whether or not to compile and link an executable program. This decision is governed by the following:

- **Generate code only** option

  When you select this option, the code generator produces code for the model, including a makefile.

- **Generate makefile** option

  When you clear this option, the code generator does not produce a makefile for the model. You must specify post code generation processing, including compilation and linking, as a user-defined command, as explained in "Customize Post-Code-Generation Build Processing" on page 70-14.

- Makefile-only target

  The Microsoft Visual C++ Project Makefile versions of the `grt` and Embedded Coder target configurations generate a Visual C++ project makefile (*`model`*`.mak`). To build an executable, you must open *`model`*`.mak` in the Visual C++ IDE and compile and link the model code.

- `HOST` template makefile variable

  The template makefile variable `HOST` identifies the type of system upon which your executable is intended to run. The variable can be set to one of three possible values: `PC`, `UNIX`, or `ANY`.

  By default, `HOST` is set to `UNIX` in template makefiles designed for use with The Open Group UNIX platforms (such as `grt_unix.tmf`), and to `PC` in the template makefiles designed for use with development systems for the PC (such as `grt_vc.tmf`).

  If the Simulink software is running on the same type of system as that specified by the `HOST` variable, then the executable is built. Otherwise,

  - If `HOST = ANY`, an executable is still built. This option is useful when you want to cross-compile a program for a system other than the one the Simulink software is running on.
  - Otherwise, processing stops after generating the model code and the makefile; the following message is displayed on the MATLAB command line.

```
### Make will not be invoked - template makefile is for a different host
```

- `TGT_FCN_LIB` template makefile variable

  The template makefile variable `TGT_FCN_LIB` specifies compiler command line options. The line in the makefile is `TGT_FCN_LIB = |>TGT_FCN_LIB<|`. Use this token in a makefile conditional statement to specify a standard math library as a compiler option. Possible `|>TGT_FCN_LIB<|` token values are:

Value	Generates Calls To
Name of custom CRL	ISO®/IEC 9899:1990 C (`ANSI_C`) standard math library
`ISO_C`	ISO/IEC 9899:1999 C standard math library
`ISO_C++`	ISO/IEC 14882:2003 C++ standard math library
`GNU`	GNU extensions to the ISO/IEC 9899:1999 C standard math library

Cross-Compile Code Generated on Microsoft Windows

If you need to generate code with the code generator on a Microsoft Windows system but compile the generated code on a different supported platform, you can do so by modifying your TMF and model configuration parameters. For example, you would need to do this if you develop applications with the MATLAB and Simulink products on a Windows system, but you run your generated code on a Linux system.

To set up a cross-compilation development environment, do the following (here a Linux system is the destination platform):

1. On your Windows system, copy the UNIX TMF for your target to a local folder. This will be your working folder for initiating code generation. For example, you might copy the file *matlabroot*`/rtw/c/grt/grt_unix.tmf` to `D:/work/my_grt_unix.tmf`.
2. Make the following changes to your copy of the TMF:
   - Add the following line near the `SYS_TARGET_FILE =` line:

     ```
     MAKEFILE_FILESEP = /
     ```
   - Search for the line `'ifeq ($(OPT_OPTS),$(DEFAULT_OPT_OPTS))'` and, for each occurrence, remove the conditional logic and retain only the `'else'` code. That is, remove everything from the `'if'` to the `'else'`, inclusive, as well as the closing `'endif'`. Only the lines from the `'else'` portion should remain. This forces the run-time libraries to build for a Linux system.
3. Open your model and make the following changes in the **Code Generation** pane of the Configuration Parameters dialog:
   - Specify the name of your new TMF in the "Template makefile" (Simulink Coder) text box (for example, `my_grt_unix.tmf`).
   - Select **Generate code only** and click **Apply**.
4. Generate the code.
5. If the build folder (folder from which the model was built) is not already Linux accessible, copy it to a Linux accessible path. For example, if your build folder for the generated code was `D:\work\mymodel_grt_rtw`, copy that entire folder tree to a path such as `/home/user/mymodel_grt_rtw`.
6. If the MATLAB folder tree on the Windows system is Linux accessible, skip this step. Otherwise, copy the include and source folders to a Linux accessible drive partition, for example, `/home/user/myinstall`. These folders appear in the makefile after

`MATLAB_INCLUDES =` and `ADD_INCLUDES =` and can be found by searching for `$(MATLAB_ROOT)`. Paths that contain `$(MATLAB_ROOT)` must be copied. Here is an example list (your list will vary depending on your model):

```
$(MATLAB_ROOT)/rtw/c/grt
$(MATLAB_ROOT)/extern/include
$(MATLAB_ROOT)/simulink/include
$(MATLAB_ROOT)/rtw/c/src
$(MATLAB_ROOT)/rtw/c/tools
```

Additionally, paths containing `$(MATLAB_ROOT)` in the build rules (lines with `%.o :`) must be copied. For example, based on the build rule

```
%.o : $(MATLAB_ROOT)/rtw/c/src/ext_mode/tcpip/%.c
```

the following folder should be copied:

```
$(MATLAB_ROOT)/rtw/c/src/ext_mode/tcpip
```

Note: The path hierarchy relative to the MATLAB root must be maintained. For example, `c:\MATLAB\rtw\c\tools\*` would be copied to `/home/user/mlroot/rtw/c/tools/*`.

For some blocksets, it is easiest to copy a higher-level folder that includes the subfolders listed in the makefile. For example, the DSP System Toolbox product requires the following folders to be copied:

```
$(MATLAB_ROOT)/toolbox/dspblks
$(MATLAB_ROOT)/toolbox/rtw/dspblks
```

---

7 Make the following changes to the generated makefile:

- Set both `MATLAB_ROOT` and `ALT_MATLAB_ROOT` equal to the Linux accessible path to *`matlabroot`* (for example, `home/user/myinstall`).
- Set `COMPUTER` to the computer value for your platform, such as `GLNX86`. Enter `help computer` in the MATLAB Command Window for a list of computer values.
- In the `ADD_INCLUDES` list, change the build folder (designating the location of the generated code on the Windows system) and parent folders to Linux accessible include folders. For example, change `D:\work\mymodel_grt_rtw\` to `/home/user/mymodel_grt_rtw`.

  Additionally, if *`matlabroot`* is a UNC path, such as `\\my-server\myapps\matlab`, replace the hard-coded MATLAB root with `$(MATLAB_ROOT)`.

8 From a Linux shell, compile the code you generated on the Windows system. You can do this by running the generated *`model`*`.bat` file or by typing the make command line as it appears in the `.bat` file.

Note: If errors occur during makefile execution, you may need to run the `dos2unix` utility on the makefile (for example, `dos2unix mymodel.mk`).

Related Examples

- Use packNGo to Relocate Code to Another Development Environment (Simulink Coder)

Control Library Location and Naming During Build

Use the `TargetPreCompLibLocation` and `TargetLibSuffix` configuration parameters to control values in generated makefiles during model builds when you use the toolchain approach or the template makefile approach.

In this section...

Library Control Parameters

Use the library control parameters to:

- Specify the location of precompiled libraries, such as blockset libraries or the Simulink Coder block library. Typically, a target has cross-compiled versions of these libraries and places them in a target-specific folder.
- Control the suffix applied to library file names (for example, `_target.a` or `_target.lib`).

Targets can set the parameters inside the system target file (STF) select callback. For example:

```
function mytarget_select_callback_handler(varargin)
    hDig=varargin{1};
    hSrc=varargin{2};
    slConfigUISetVal(hDig, hSrc, 'TargetPreCompLibLocation',...
    'c:\mytarget\precomplibs');
    slConfigUISetVal(hDig, hSrc, 'TargetLibSuffix',...
    '_target.library');
```

The TMF has corresponding expansion tokens:

```
    |>EXPAND_LIBRARY_LOCATION<|
    |>EXPAND_LIBRARY_SUFFIX<|
```

Alternatively, you can use a call to the `set_param` function. For example:

```
set_param(model,'TargetPreCompLibLocation',...
```

```
'c:\mytarget\precomplibs');
```

Note: If your model contains referenced models, you can use the make option `USE_MDLREF_LIBPATHS` to control whether libraries used by the referenced models are copied to the parent model's build folder. For more information, see "Control the Location of Model Reference Libraries" on page 70-10.

Using TargetLibSuffix with the Toolchain Approach

With `TargetLibSuffix`, you specify a suffix followed by an extension. For example: *`suffix.extension`*

However, when you use the toolchain approach, only the suffix provided by `TargetLibSuffix` is honored. The extension that the `TargetLibSuffix` provides is not honored because the toolchain approach provides a different extension.

For example, with the target makefile approach, the final binary name is composed of the *`modelname`*, the *`suffix`*, and the *`extension`* provided by `TargetLibSuffix`:

```
modelname+suffix.extension_from_TargetLibSuffix
```

With the toolchain approach, the final binary name is composed of the *`modelname`*, the *`suffix`*, and the *`extension`* provided by the toolchain approach:

```
model+suffix.extension_from_toolchain_approach
```

The extension that the toolchain approach uses comes from the file extension of the static library that the build tool creates. To get this information:

Create the toolchain object. For example, enter:

```
tc = ToolchainInfo used in the model
```

Get the build tool name. For example, enter:

```
tool = tc.getBuildTool('C Compiler');
```
or
```
tool = tc.getBuildTool('C++ Compiler');
```
Get the extension. For example, enter:

```
extension_from_ToolchainInfo = tool.getFileExtension('Static Library')
```

Note: Note: If you do not set the `TargetLibSuffix` parameter, template makefile and toolchain approaches behave identically. See "Customize Library File Suffix and File Type" (Simulink Coder).

Specify the Location of Precompiled Libraries

Use the `TargetPreCompLibLocation` configuration parameter to:

- Override the precompiled library location specified in the `rtwmakecfg.m` file (see "Use rtwmakecfg.m API to Customize Generated Makefiles" on page 70-26 for details)
- Precompile and distribute target-specific versions of product libraries (for example, the DSP System Toolbox product)

For a precompiled library, such as a blockset library or the Simulink Coder block library, the location specified in `rtwmakecfg.m` is typically a location specific to the blockset or the Simulink Coder product. The code generator expects that the library exists in this location and links against the library during builds.

However, for some applications, such as custom targets, it is preferable to locate the precompiled libraries in a target-specific or other alternate location rather than in the location specified in `rtwmakecfg.m`. For a custom target, the code generator expects that the target-specific cross-compiler creates the library, and you place the library in the target-specific location. Libraries supported by the target should be compiled and placed in the target-specific location so they can be used during the build process.

You can set up the `TargetPreCompLibLocation` parameter in its select callback. The path that you specify for the parameter must be a fully qualified, absolute path to the library location. Relative paths are not supported. For example:

```
slConfigUISetVal(hDlg, hSrc, 'TargetPreCompLibLocation',...
'c:\mytarget\precomplibs');
```

Alternatively, you set the parameter with a call to the `set_param` function. For example:

```
set_param(model,'TargetPreCompLibLocation',...
'c:\mytarget\precomplibs');
```

During makefile generation, the build process replaces the tokens with the location from the `rtwmakecfg.m` file. For example, if the library name in the `rtwmakecfg.m` file is `'rtwlib'`, the template makefile build approach expands the token from:

```
LIBS += |>EXPAND_LIBRARY_LOCATION<|\|>EXPAND_LIBRARY_NAME<|\
```

```
_target.library
```

to:

```
LIBS += c:\mytarget\precomplibs\rtwlib_target.library
```

By default, `TargetPreCompLibLocation` is an empty character vector. The build process uses the location in `rtwmakecfg.m` for the token replacement.

Control the Location of Model Reference Libraries

On platforms other than the Apple Macintosh platform, when building a model that uses referenced models, the default build process includes:

- Copy libraries that the referenced models uses to the parent model's build folder.
- Assign the file names of the libraries to `MODELREF_LINK_LIBS` in the generated makefile.

For example, if a model includes a referenced model `sub`, the build process assigns the library name `sub_rtwlib.lib` to `MODELREF_LINK_LIBS`. The build process copies the library file to the parent model's build folder. This definition is then used in the final link line, which links the library into the final product (usually an executable). This technique minimizes the length of the link line.

On the Macintosh platform, and optionally on other platforms, the build process includes:

- No copying of libraries that the referenced models uses to the parent model's build folder.
- Assign the relative paths and file names of the libraries to `MODELREF_LINK_LIBS` in the generated makefile.

When using this technique, the build process assigns a relative path such as `../slprj/grt/sub/sub_rtwlib.lib` to `MODELREF_LINK_LIBS`. The build process uses the path to gain access to the library file at link time.

To change to the nondefault behavior on platforms other than the Macintosh platform, select the **Configuration Parameters** > **Code Generation** > **Make command** field. Enter:

```
make_rtw USE_MDLREF_LIBPATHS=1
```

If you specify other Make command arguments, such as `OPTS="-g"`, the order in which you specify the multiple arguments does not matter.

To return to the default behavior, set USE_MDLREF_LIBPATHS to 0, or remove it.

Control the Suffix Applied to Library File Names

Use the TargetLibSuffix configuration parameter to control the suffix applied to library names (for example, _target.lib or _target.a). The specified suffix scheme must include a period (.). You can apply TargetLibSuffix to the following libraries:

- Libraries on which a target depends, as specified in the rtwmakecfg.m API. You can use TargetLibSuffix to change the suffix of both precompiled and non-precompiled libraries configured from the rtwmakecfg API. For details, see "Use rtwmakecfg.m API to Customize Generated Makefiles" on page 70-26.

  In this case, a target can set the parameter in its select callback. For example:

  ```
  slConfigUISetVal(hDlg, hSrc, 'TargetLibSuffix',...
  '_target.library');
  ```

  Alternatively, you can use a call to the set_param function. For example:

  ```
  set_param(model,'TargetLibSuffix','_target.library');
  ```

  During the TMF-to-makefile conversion, the build process replaces the token |>EXPAND_LIBRARY_SUFFIX<| with the specified suffix. For example, if the library name specified in the rtwmakecfg.m file is 'rtwlib', the TMF expands from:

  ```
  LIBS += |>EXPAND_LIBRARY_LOCATION<|\|>EXPAND_LIBRARY_NAME<|\
  |>EXPAND_LIBRARY_SUFFIX<|
  ```

  to:

  ```
  LIBS += c:\mytarget\precomplibs\rtwlib_target.library
  ```

  By default, TargetLibSuffix is set to an empty character vector. In this case, the build process replaces the token |>EXPAND_LIBRARY_SUFFIX<| with an empty character vector.

- Shared utility library and the model libraries created with model reference. For these cases, associated makefile variables do not require the |>EXPAND_LIBRARY_SUFFIX<| token. Instead, the build process includes TargetLibSuffix implicitly. For example, for a top model named topmodel with referenced models named refmodel1 and refmodel2, the top model's TMF is expanded from:

  ```
  SHARED_LIB          = |>SHARED_LIB<|
  ```

```
MODELLIB           = |>MODELLIB<|
MODELREF_LINK_LIBS = |>MODELREF_LINK_LIBS<|
```

to:

```
SHARED_LIB         = \
..\slprj\ert\_sharedutils\rtwshared_target.library
MODELLIB           = topmodellib_target.library
MODELREF_LINK_LIBS = \
refmodel1_rtwlib_target.library refmodel2_rtwlib_target.library
```

By default, the `TargetLibSuffix` parameter is an empty character vector. In this case, the build process chooses a default suffix for these three tokens using a file extension of `.lib` on Windows hosts and `.a` on UNIX hosts. (For model reference libraries, the default suffix additionally includes `_rtwlib`.) For example, on a Windows host, the expanded makefile values are:

```
SHARED_LIB         = ..\slprj\ert\_sharedutils\rtwshared.lib
MODELLIB           = topmodellib.lib
MODELREF_LINK_LIBS = refmodel1_rtwlib.lib refmodel2_rtwlib.lib
```

Recompile Precompiled Libraries

You can recompile precompiled libraries included as part of the code generator, such as `rtwlib` or `dsplib`, by using a supplied MATLAB function, `rtw_precompile_libs`. You might consider doing this if you need to customize compiler settings for various platforms or environments. For details on using `rtw_precompile_libs`, see "Precompile S-Function Libraries" on page 39-47.

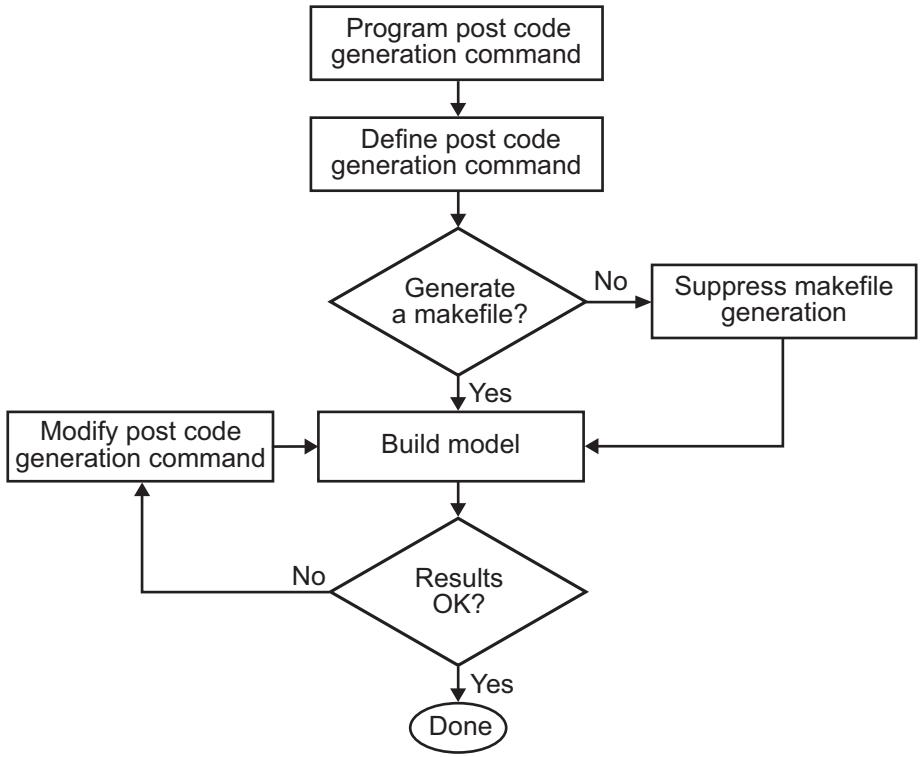
Customize Post-Code-Generation Build Processing

The code generator provides a set of tools, including a build information object, you can use to customize build processing that occurs after code generation. You might use such customizations for target development or the integration of third-party tools into your application development environment.

In this section...

Workflow for Setting Up Customizations

The following figure and the steps that follow show the general workflow for setting up post-code-generation customizations.



1. Program the post code generation command.
2. Define the post code generation command.
3. Suppress makefile generation, if applicable.
4. Build the model.
5. Modify the command and rebuild the model until the build results are acceptable.

Build Information Object

At the start of a model build, the build process logs the following build option and dependency information to a temporary build information object:

- Compiler options
- Preprocessor identifier definitions
- Linker options

- Source files and paths
- Include files and paths
- Precompiled external libraries

You can retrieve information from and add information to this object by using an extensive set of functions. For a list of available functions and detailed function descriptions, see "Build Process Customization" (Simulink Coder). "Program a Post Code Generation Command" on page 70-16 explains how to use the functions to control post code generation build processing.

Program a Post Code Generation Command

For certain applications, you might want to control aspects of the build process after the code generation. For example, you might do this if you develop your own target, or you want to apply an analysis tool to the generated code before continuing with the build process. You can apply this level of control to the build process by programming and then defining a post code generation command.

A post code generation command is a MATLAB language file that typically calls functions that get data from or add data to the model's build information object. You can program the command as a script or function.

If You Program the Command as a...	Then the...
Script	Script can gain access to the model name and the build information directly
Function	Function can pass the model name and the build information as arguments

If your post code generation command calls user-defined functions, make sure the functions are on the MATLAB path. If the build process cannot find a function you use in your command, the build process errors out.

You can then call a combination of build information functions, as listed in "Build Process Customization" (Simulink Coder), to customize the model's post code generation build processing.

The following example shows a fragment of a post code generation command that gets the file names and paths of the source and include files generated for a model for analysis.

```
function analyzegencode(buildInfo)
% Get the names and paths of source and include files
% generated for the model and then analyze them.

% buildInfo - build information for my model.

% Define cell array to hold data.
MyBuildInfo={};

% Get source file information.
MyBuildInfo.srcfiles=getSourceFiles(buildInfo, true, true);
MyBuildInfo.srcpaths=getSourcePaths(buildInfo, true);

% Get include (header) file information.
MyBuildInfo.incfiles=getIncludeFiles(buildInfo, true, true);
MyBuildInfo.incpaths=getIncludePaths(buildInfo, true);

% Analyze generated code.
.
.
.
```

Define a Post Code Generation Command

After you program a post code generation command, you need to inform the build process that the command exists and to add it to the model's build processing. You do this by defining the command with the `PostCodeGenCommand` model configuration parameter. When you define a post code generation command, the build process evaluates the command after generating and writing the model's code to disk and before generating a makefile.

As the following syntax lines show, the arguments that you specify when setting the configuration parameter varies depending on whether you program the command as a script, function, or set of functions.

Note: When defining the command as a function, you can specify an arbitrary number of input arguments. To pass the model's name and build information to the function, specify identifiers `modelName` and `buildInfo` as arguments.

Script

```
set_param(model, 'PostCodeGenCommand',...
'pcgScriptName');
```

Function

```
set_param(model, 'PostCodeGenCommand',...
 'pcgFunctionName(modelName)');
```

Multiple Functions

```
pcgFunctions=...
'pcgFunction1Name(modelName);...
pcgFunction2Name(buildInfo)';
set_param(model, 'PostCodeGenCommand',...
 pcgFunctions);
```

The following call to `set_param` defines `PostCodGenCommand` to evaluate the function `analyzegencode`.

```
set_param(model, 'PostCodeGenCommand',...
'analyzegencode(buildInfo)');
```

Customize Build Process with PostCodeGenCommand and Relocate Generated Code to an External Environment

This example shows how to use the Build Information API and the **Post Code Generation Command** parameter, `PostCodeGenCommand`.

The `PostCodeGenCommand` parameter value is `rtwdemo_buildinfo_data`. This value directs the build process to invoke the function after code generation.

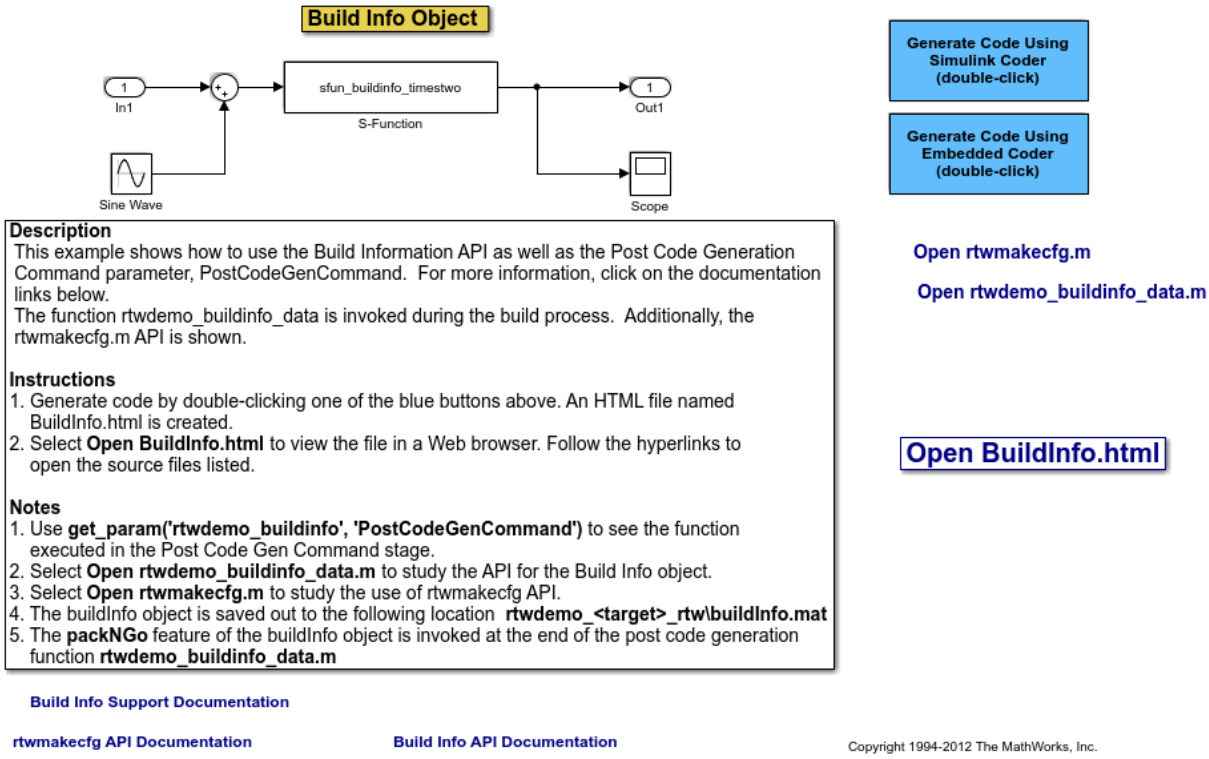
The example also demonstrates how to use the `rtwmakecfg.m` API.

For more information, click on the documentation links in the model.

Open Example Model

Open the example model `rtwdemo_buildinfo`.

```
open_system('rtwdemo_buildinfo');
```



Generate Code from Model

Double-click on the **Generate Code Using Simulink Coder** button to generate code for the GRT target.

Or, if Embedded Coder is installed, double-click on the **Generate Code Using Embedded Coder** button to generate code for the ERT target.

The build process generates a `BuildInfo.html` file to document the build information object.

Examine the Build Process Customizations and Output

Use the links in the model to examine the build process customizations and the post code generation query of the build information object.

To view the `BuildInfo.html` file in a Web browser, click on **Open BuildInfo.html**.

The example uses the `PostCodeGenCommand` parameter of the model to generate the html file from the build information object. The file provides hyperlinks to open the source files (generated code) from the model. To view the `PostCodeGenCommand` parameter value, type:

```
get_param('rtwdemo_buildinfo','PostCodeGenCommand');
```

This value indicates a function to execute in the **Post Code Gen Command** stage.

```
rtwdemo_buildinfo_data(buildInfo);
```

To study how the example uses the `rtwmakecfg` API, click on **Open rtwmakecfg.m** or type:

```
edit rtwmakecfg.m;
```

To study the API for the `buildInfo.mat` object, click on **Open rtwdemo_buildinfo_data.m** or type:

```
edit rtwdemo_buildinfo_data.m;
```

The `buildInfo.mat` object is available at:

```
rtwdemo_<target>_rtw\buildInfo.mat
```

At the end of the `rtwdemo_buildinfo_data.m` post code generation function, the function invokes `packNGo` to package the source and objects identified in the `buildInfo` object for relocation.

Further Study Topics

- “Build Process Customization” (Simulink Coder)
- “Customize Post-Code-Generation Build Processing” (Simulink Coder)
- “Use rtwmakecfg.m API to Customize Generated Makefiles” (Simulink Coder)
- “Relocate Code to Another Development Environment” (Simulink Coder)

Suppress Makefile Generation

The code generator provides the ability to suppress makefile generation during the build process. For example, you might do this to integrate tools into the build process that are not driven by makefiles.

To instruct the code generator to not produce a makefile, do one of the following:

- Clear the **Generate makefile** option on the **Code Generation** pane of the Configuration Parameters dialog box.
- Set the value of the configuration parameter `GenerateMakefile` to `off`.

When you suppress makefile generation,

- You cannot explicitly specify a make command or template makefile.
- You must specify your own instructions for a post code generation processing, including compilation and linking, in a post code generation command as explained in “Program a Post Code Generation Command” on page 70-16 and “Define a Post Code Generation Command” on page 70-17.

Configure Generated Code with TLC

In this section...

About Configuring Generated Code with TLC

You can use the Target Language Compiler (TLC) to fine tune your generated code. TLC supports extended code generation variables and options in addition to parameters available on the **Code Generation** pane on the Configuration Parameters dialog box. There are two ways to set TLC variables and options, as described in this section.

Note: You should not customize TLC files in the folder *`matlabroot`*`/rtw/c/tlc` even though the capability exists to do so. Such TLC customizations might not be applied during the code generation process and can lead to unpredictable results.

Assigning Target Language Compiler Variables

The `%assign` statement lets you assign a value to a TLC variable, as in

```
%assign MaxStackSize = 4096
```

This is also known as creating a *parameter name/parameter value pair*.

For a description of the `%assign` statement see "Target Language Compiler Directives" (Simulink Coder). You should write your `%assign` statements in the `Configure RTW code generation settings` section of the system target file.

The following table lists the code generation variables you can set with the `%assign` statement.

Target Language Compiler Optional Variables

Variable	Description
`MaxStackSize=`*`N`*	When the **Enable local block outputs** check box is selected, the total allocation size of local variables that

Variable	Description
	are declared by block outputs in the model cannot exceed `MaxStackSize` (in bytes). `MaxStackSize` can be a 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, such as from a Stateflow diagram or MATLAB Function block, 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, do a target-specific measurement by using run-time (empirical) analysis or static (code path) analysis with object code.
`MaxStackVariableSize=`*`N`*	When the **Enable local block outputs** check box is selected, this limits the size of a 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.
`WarnNonSaturatedBlocks=`*`value`*	Flag to control display of overflow warnings for blocks that have saturation capability, but have it turned off (unchecked) in their dialog. These are the options: • `0` — Warning is not displayed. • `1` — Displays one warning for the model during code generation • `2` — Displays one warning that contains a list of offending blocks

Set Target Language Compiler Options

You can specify TLC command line options for code generation using the model parameter `TLCOptions` in a `set_param` function call. For information about these options, see “Specify TLC for Code Generation” (Simulink Coder) and “Configure TLC” (Simulink Coder).

Use makecfg to Customize Generated Makefiles for S-Functions

With the toolchain and template makefile approach for building code, you can customize generated makefiles for S-functions. Through the customization, you can specify additional items for the S-function build process:

- Source files and folders
- Include files and folders
- Library names
- Preprocessor macro definitions
- Compiler flags
- Link objects

1. To customize the generated makefile:
   - For all S-functions in the build folder (Simulink Coder), create a `makecfg.m` file.
   - For a specific S-function in the build folder, create a *`specificSFunction`*`_makecfg.m` file.
2. In the file that you create, use `RTW.BuildInfo (Simulink Coder)` functions to specify additional items for the S-function build process. For example, you can use:
   - `addCompileFlags` to specify compiler options.
   - `addDefines` to specify preprocessor macro definitions.
3. Save the created file in the build folder.

After code generation, in the build folder, the code generator searches for `makecfg.m` and *`specificSFunction`*`_makecfg.m` files. If the files are present in the build folder, the code generator uses these files to customize the generated makefile, *`model`*`.mk`.

For example, consider a build folder that contains `signalConvert.mexa64` (S-function binary file) and `signalConvert.tlc` (inlined S-function implementation) after the TLC phase (Simulink Coder) of the build process. The S-function requires an additional source code file, `filterV1.c`, which is located in `anotherFolder`. You can create a file, `signalConvert_makecfg.m`, that uses `RTW.BuildInfo` functions to specify `filterV1.c` for the build process.

```
function signalConvert_makecfg(objBuildInfo)
```

```
absolute = fullfile('$(START_DIR)', 'anotherFolder');

addIncludePaths(objBuildInfo, absolute);
addSourcePaths(objBuildInfo, absolute);
addSourceFiles(objBuildInfo,'filterV1.c');
```

Related Examples

- “Build Process Workflow for a Real-Time STF” (Simulink Coder)
- “Choose and Configure Build Process” (Simulink Coder)
- “Import Calls to External Code into Generated Code with Legacy Code Tool” (Simulink Coder)
- “Use rtwmakecfg.m API to Customize Generated Makefiles” (Simulink Coder)

Use rtwmakecfg.m API to Customize Generated Makefiles

Both the toolchain approach and the template makefile approach for builds let you add the following items to generated makefiles:

- Source folders
- Include folders
- Library names
- Module objects

About the rtwmakecfg Function

Using an `rtwmakecfg` function, you add this information to the makefile during the build operation for S-functions. The `rtwmakecfg` function is particularly useful when specifying added sources and libraries to build a model that contains one or more of your S-function blocks.

To add information pertaining to an S-function to the makefile:

1. Create the MATLAB language `rtwmakecfg` function in the `rtwmakecfg.m` file. The code generator associates this file with your S-function based on its folder location. “Create the rtwmakecfg Function” on page 70-26 describes the requirements for the `rtwmakecfg` function and the data it returns.
2. If you are using the template makefile approach, modify your target's TMF such that it supports macro expansion for the information that the `rtwmakecfg` function returns. “Modify the Template Makefile for rtwmakecfg” on page 70-29 describes the required modifications. If you are using the toolchain approach, the information that the rtwmakecfg function returns is used by the generated makefile; no further configuration is required.

After the TLC phase of the build process, when generating a makefile, the code generator searches for an `rtwmakecfg.m` file in the folder that contains the S-function MEX file. If it finds the file, the build process calls the `rtwmakecfg` function.

Create the rtwmakecfg Function

Create the `rtwmakecfg.m` file containing the `rtwmakecfg` function in the same folder as your S-function component (a MEX-file with a platform-dependent extension, such as

`.mexext` on Microsoft Windows systems). The function must return a structured array that contains these fields.

Field	Description
`makeInfo.includePath`	A cell array that specifies additional include folder names, organized as a row vector. The build process expands the folder names into include instructions in the generated makefile.
`makeInfo.sourcePath`	A cell array that specifies additional source folder names, organized as a row vector. You must include the folder names of files entered into the **S-function modules** field on the S-Function Block Parameters dialog box or into the block's `SFunctionModules` parameter if they are not in the same folder as the S-function. The build process expands the folder names into make rules in the generated makefile.
`makeInfo.sources`	A cell array that specifies additional source file names (C or C ++), organized as a row vector. Do not include the name of the S-function or any files entered into the **S-function modules** field on the S-Function Block Parameters dialog box or into the block's `SFunctionModules` parameter. The build process expands the file names into make variables that contain the source files. Specify only file names (with extension). Specify path information with the `sourcePath` field.
`makeInfo.linkLibsObjs`	A cell array that specifies additional, fully qualified paths to object or library files against which the generated code links. The build process does not compile the specified objects and libraries. However, it includes them when linking the final executable. This inclusion can be useful for incorporating libraries that you do not want the build process to recompile or for which the source files are not available. You can also use this element to incorporate source files from languages other than C and C++. This is possible if you first create a C compatible object file or library outside of the build process.
`makeInfo.precompile`	A Boolean flag that indicates whether the libraries specified in the `rtwmakecfg.m` file exist in a specified location (`precompile==1`) or if you must create the libraries in the build folder during the build process (`precompile==0`).

Field	Description
`makeInfo.library`	A structure array that specifies additional run-time libraries and module objects, organized as a row vector. The build process expands the information into make rules in the generated makefile. For a list of the library fields, see the next table.

The `makeInfo.library` field consists of the following elements.

Element	Description
`makeInfo.library(n).Name`	A character array that specifies the name of the library (without an extension).
`makeInfo.library(n).Location`	A character array that specifies the folder in which the library is located when precompiled. For more information, see the description of `makeInfo.precompile` in the preceding table. A target can use the `TargetPreCompLibLocation` parameter to override this value. See "Specify the Location of Precompiled Libraries" (Simulink Coder).
`makeInfo.library(n).Modules`	A cell array that specifies the C or C++ source file base names (without an extension) that comprise the library. Do not include the file extension. The makefile appends the object extension.

Note: The `makeInfo.library` field must fully specify each library and how to build it. The modules list in the `makeInfo.library(n).Modules` element cannot be empty. If you need to specify a link-only library, use the `makeInfo.linkLibsObjs` field instead.

Example:

```
disp(['Running rtwmakecfg from folder: ',pwd]);
   makeInfo.includePath = { fullfile(pwd, 'somedir2') };
   makeInfo.sourcePath = {fullfile(pwd, 'somedir2'), fullfile(pwd, 'somedir3')};
   makeInfo.sources  = { 'src1.c', 'src2.cpp'};
   makeInfo.linkLibsObjs = { fullfile(pwd, 'somedir3', 'src3.object'),...
                             fullfile(pwd, 'somedir4', 'mylib.library')};
   makeInfo.precompile = 1;
   makeInfo.library(1).Name     = 'myprecompiledlib';
   makeInfo.library(1).Location = fullfile(pwd,'somdir2','lib');
   makeInfo.library(1).Modules  = {'srcfile1' 'srcfile2' 'srcfile3' };
```

Note: If a path that you specify in the `rtwmakecfg.m` API contains spaces, the build process does not convert the path to its nonspace equivalent. If the build environments you intend to support do not support spaces in paths, refer to "Enable Build Process for Folder Names with Spaces" (Simulink Coder).

Modify the Template Makefile for rtwmakecfg

To expand the information that an `rtwmakecfg` function generates, modify the following sections of your target's TMF:

- `Include Path`
- `C Flags` and/or `Additional Libraries`
- `Rules`

It is possible that these TMF code examples do not apply to your make utility. For additional examples, see the GRT or ERT TMFs located in *`matlabroot`*`/rtw/c/grt` (open) or *`matlabroot`*`/rtw/c/ert` (open).

Add Folder Names to the Makefile Include Path

The following TMF code example adds folder names to the include path in the generated makefile:

```
ADD_INCLUDES = \
|>START_EXPAND_INCLUDES<|   -I|>EXPAND_DIR_NAME<| \
|>END_EXPAND_INCLUDES<|
```

Additionally, the `ADD_INCLUDES` macro must be added to the `INCLUDES` line.

```
INCLUDES = -I. -I.. $(MATLAB_INCLUDES) $(ADD_INCLUDES) $(USER_INCLUDES)
```

Add Library Names to the Makefile

The following TMF code example adds library names to the generated makefile.

```
LIBS =
|>START_PRECOMP_LIBRARIES<|
LIBS += |>EXPAND_LIBRARY_NAME<|.a |>END_PRECOMP_LIBRARIES<|
|>START_EXPAND_LIBRARIES<|
LIBS += |>EXPAND_LIBRARY_NAME<|.a |>END_EXPAND_LIBRARIES<|
```

For more information, see "Control Library Location and Naming During Build" on page 70-7.

Add Rules to the Makefile

The TMF code example adds rules to the generated makefile.

```
|>START_EXPAND_RULES<|
$(BLD)/%.o: |>EXPAND_DIR_NAME<|/%.c $(SRC)/$(MAKEFILE) rtw_proj.tmw
    @$(BLANK)
    @echo ### "|>EXPAND_DIR_NAME<|\$*.c"
    $(CC) $(CFLAGS) $(APP_CFLAGS) -o $(BLD)$(DIRCHAR)$*.o \
    |>EXPAND_DIR_NAME<|$(DIRCHAR)$*.c > $(BLD)$(DIRCHAR)$*.lst
|>END_EXPAND_RULES<|

|>START_EXPAND_LIBRARIES<|MODULES_|>EXPAND_LIBRARY_NAME<| = \
|>START_EXPAND_MODULES<|    |>EXPAND_MODULE_NAME<|.o \
|>END_EXPAND_MODULES<|

|>EXPAND_LIBRARY_NAME<|.a : $(MAKEFILE) rtw_proj.tmw
$(MODULES_|>EXPAND_LIBRARY_NAME<|:%.o=$(BLD)/%.o)
    @$(BLANK)
    @echo ### Creating $@
    $(AR) -r $@ $(MODULES_|>EXPAND_LIBRARY_NAME<|:%.o=$(BLD)/%.o)
|>END_EXPAND_LIBRARIES<|

|>START_PRECOMP_LIBRARIES<|MODULES_|>EXPAND_LIBRARY_NAME<| = \
|>START_EXPAND_MODULES<|    |>EXPAND_MODULE_NAME<|.o \
|>END_EXPAND_MODULES<|

|>EXPAND_LIBRARY_NAME<|.a : $(MAKEFILE) rtw_proj.tmw
$(MODULES_|>EXPAND_LIBRARY_NAME<|:%.o=$(BLD)/%.o)
    @$(BLANK)
    @echo ### Creating $@
    $(AR) -r $@ $(MODULES_|>EXPAND_LIBRARY_NAME<|:%.o=$(BLD)/%.o)
|>END_PRECOMP_LIBRARIES<|
```

Customize Build Process with STF_make_rtw_hook File

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

In this section...

The STF_make_rtw_hook File

You can modify hook methods in a file 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 file implements a function, *STF*`_make_rtw_hook`, that dispatches to a specific action, depending on the `hookMethod` argument passed in.

The build process calls *STF*`_make_rtw_hook`, passing in the`hookMethod` argument and other arguments. You implement only those hook methods that your build process requires.

If your model contains reference models, you can implement an *STF*`_make_rtw_hook.m` for each reference model as required. The build process calls each *STF*`_make_rtw_hook` for reference models, processing these files recursively (in dependency order).

Conventions for Using the STF_make_rtw_hook File

For the build process to call the *STF*`_make_rtw_hook`, check that the following conditions are met:

- The *STF*`_make_rtw_hook.m` file is on the MATLAB path.

- The file name is the name of your system target file (`STF`), appended to the text `_make_rtw_hook.m`. For example, if you generate code with a custom system target file `mytarget.tlc`, name your hook file `mytarget_make_rtw_hook.m`, and name the hook function implemented within the file `mytarget_make_rtw_hook`.
- The hook function implemented in the file uses the function prototype described in "STF_make_rtw_hook.m Function Prototype and Arguments" on page 70-32.

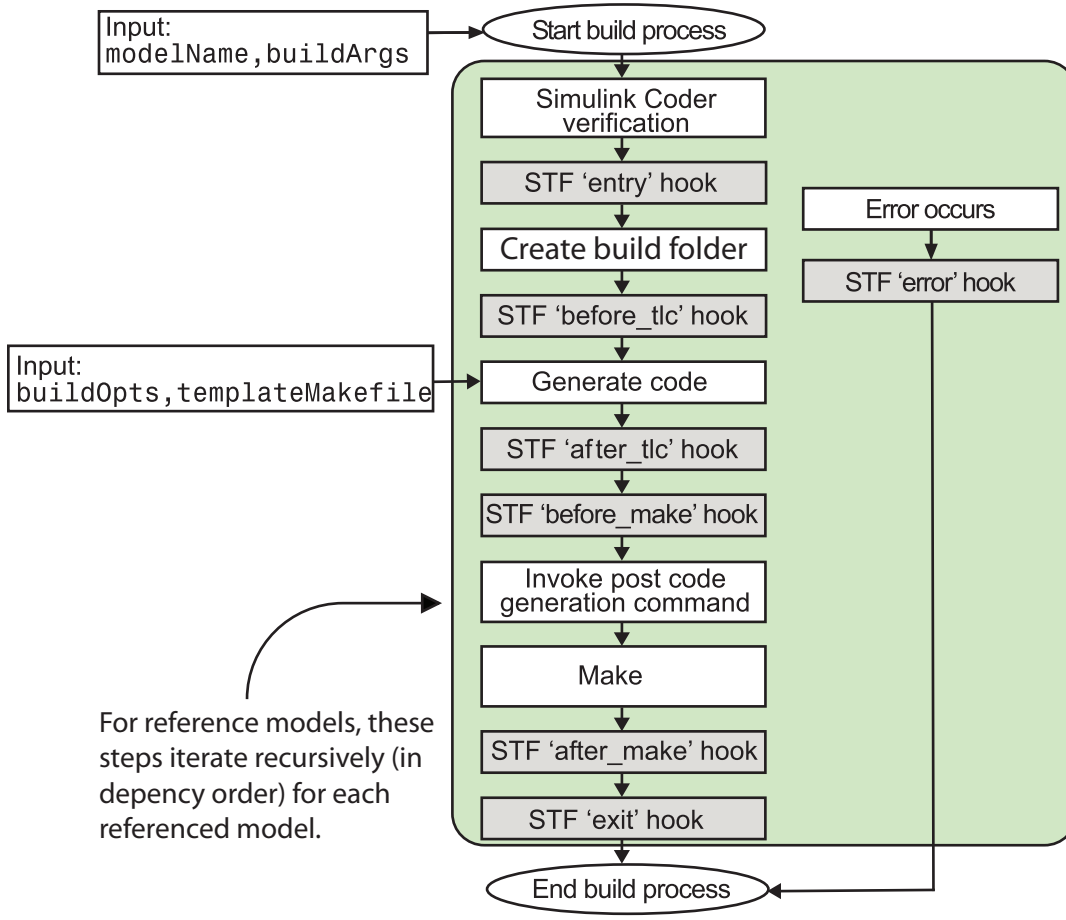
STF_make_rtw_hook.m Function Prototype and Arguments

The function prototype for *`STF`*`_make_rtw_hook` is:

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

The arguments are defined as:

- `hookMethod`: Character vector specifying the stage of build process from which the *`STF`*`_make_rtw_hook` function is called. The following flowchart 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`: Character vector 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 following list. Valid for the `'before_make'`, `'after_make'`, and `'exit'` stages only. The `buildOpts` fields include:
  - `modules`: Character vector specifying a list of additional files to compile.

  - `codeFormat`: Character vector specifying the value of the `CodeFormat` TLC variable for the target. (ERT-based targets must use the `'Embedded-C'` value for the `CodeFormat` TLC variable and use the corresponding `'ert.tlc'` value in the `rtwgensettings.DerivedFrom` field.)
  - `noninlinedSFcns`: Cell array specifying a list of noninlined S-functions in the model.
  - `compilerEnvVal`: Character vector specifying the compiler environment variable value (for example, `C:\Applications\Microsoft Visual`).
- `buildArgs`: Character vector containing the argument to `make_rtw`. When you invoke the build process, `buildArgs` is copied from the argument following `"make_rtw"` in the **Configuration Parameters**+**Code Generation**+**Make command** field.

  For example, the following make arguments from the **Make command** field

  ```
  make_rtw VAR1=0 VAR2=4
  ```

  generate the following:

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

  The `buildArgs` argument does not apply for toolchain approach builds because these builds do not allow adding make arguments to the `make_rtw` call. To provide custom definitions (for example, `VAR1=0 VAR2=4`) on the compiler command line that apply for both TMF approach and toolchain approach builds, use the **Configuration Parameters** > **Code Generation** > **Custom Code** > **Defines** field.
- `buildInfo`: The MATLAB structure that contains the model build information fields. Valid for the `'after_tlc'`, `'before_make'`, `'after_make'`, and `'exit'` stages only. For information about these fields and functions to access them, see “Build Process Customization” (Simulink Coder).

Applications for STF_make_rtw_hook.m

Here are some examples of how you might apply the *STF*_make_rtw_hook.m hook methods.

In general, you can use the 'entry' hook to initialize the build process, for example, to change or validate settings before code is generated. 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 a stage after 'entry' to obtain the path to the build folder. At the 'exit' stage, you could then locate generated code files within the build folder 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: The build process temporarily changes the MATLAB working folder to the build folder for stages 'before_make', 'after_make', 'exit', and 'error'. Your *STF*_make_rtw_hook.m file must not make incorrect assumptions about the location of the build folder. At a point after the 'entry' stage, you can obtain the path to the build folder. In the following MATLAB code example, the build folder path is returned as a character vector to the variable buildDirPath.

```
buildDirPath = rtwprivate('get_makertwsettings',gcs,'BuildDirectory');
```

Control Code Regeneration Using STF_make_rtw_hook.m

When you rebuild a model, by default, the build process performs checks to determine whether changes to the model or relevant settings require regeneration of the top model code. (For details on the criteria, see “Control Regeneration of Top Model Code” on page 40-48.) If the checks determine that top model code generation is required, the build process fully regenerates and compiles the model code. If the checks indicate that the top model generated code is current with respect to the model, and model settings do not require full regeneration, the build process omits regeneration of the top model code.

Regardless of whether the top model code is regenerated, the build process subsequently calls the build process hooks, including *STF*`_make_rtw_hook` functions and the post code generation command. The following mechanisms allow you to perform actions related to code regeneration in the *STF*`_make_rtw_hook` functions:

- To force code regeneration, use the following function call from the `'entry'` hook:

  ```
  rtw.targetNeedsCodeGen('set', true);
  ```

- In hooks from `'before_tlc'` through `'exit'`, the `buildOpts` structure passed to the hook has a Boolean field `codeWasUpToDate`. The field is set to `true` if model code was up to date and code was not regenerated, or `false` if code was not up to date and code was regenerated. You can customize hook actions based on the value of this field. For example:

  ```
  ...
  case 'before_tlc'
      if buildOpts.codeWasUpToDate
          %Perform hook actions for up to date model
      else
          %Perform hook actions for full code generation
      end
  ...
  ```

Use 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, which is located in the folder*`matlabroot`*`/toolbox/coder/embeddedcoder` (open), as follows:

1. Copy `ert_make_rtw_hook.m` to a folder in the MATLAB path. Rename it in accordance with the naming conventions described in "Conventions for Using the STF_make_rtw_hook File" on page 70-31. 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 file name.
3. Implement the hooks that you require by adding code to case statements within the `switch hookMethod` statement.

Customize Build Process with sl_customization.m

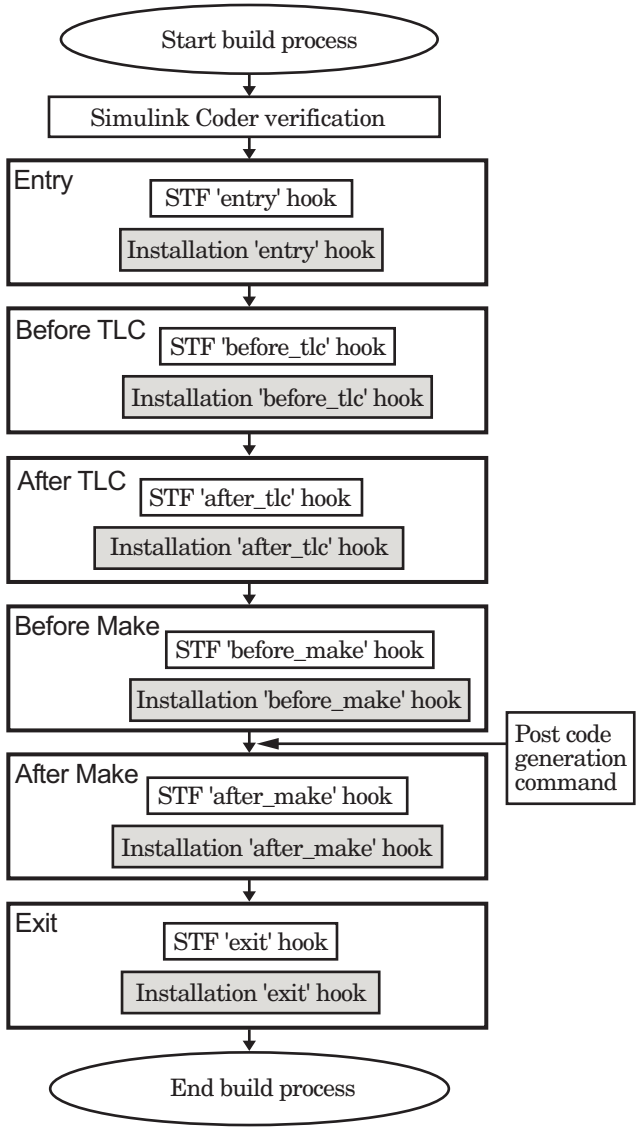
The Simulink customization file `sl_customization.m` is a mechanism that allows you to use MATLAB to customize the build process interface. The Simulink software reads the `sl_customization.m` file, if present on the MATLAB path, when it starts and the customizations specified in the file are applied to the Simulink session. For more information on the `sl_customization.m` customization file, see "Registering Customizations" (Simulink).

In this section...

The sl_customization.m File

The `sl_customization.m` file can be used to register installation-specific hook functions to be invoked during the build process. The hook functions that you register through `sl_customization.m` complement System Target File (STF) hooks (described in "Customize Build Process with STF_make_rtw_hook File" on page 70-31) and post-code generation commands (described in "Customize Post-Code-Generation Build Processing" on page 70-14).

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


Start build process
Simulink Coder 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

Register Build Process Hook Functions Using sl_customization.m

To register installation-level hook functions that will be invoked during the build process, you create a MATLAB 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 a customization manager object. For example,

```
function sl_customization(cm)
```

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

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

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

- `addUserHook(hObj, hookType, hook)`

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

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

The Simulink software reads the `sl_customization.m` file when it starts. If you subsequently change the file, you must restart the Simulink session or enter the following command in the Command Window to enable 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)

A hook script can directly access the valid variables. A hook function 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 Build Process Customizations uses the `addUserHook` method to specify installation-specific build process hooks to be invoked at the `'entry'` and `'after_tlc'` stages of the build process. For the hook function source code, see Example 2: CustomRTWEntryHook.m and Example 3: CustomRTWPostProcessHook.m.

Example 1: sl_customization.m for Build Process Customizations

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

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

% Register 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 builds. For example, if you open the ERT-based model `rtwdemo_udt`, open the **Code Generation** pane of the Configuration Parameters dialog box, and press **Ctrl+B** to initiate a build, the following messages are displayed:

```
>> rtwdemo_udt

### Starting build procedure for model: rtwdemo_udt
Custom entry hook for model 'rtwdemo_udt.'
Custom post process hook for model 'rtwdemo_udt.'
```

```
### Successful completion of build procedure for model: rtwdemo_udt
>>
```

Replace STF_rtw_info_hook Supplied Target Data

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 properties that were formerly specified in your *STF*`_rtw_info_hook` file.

For backward compatibility, existing *STF*`_rtw_info_hook` files are available. However, you should convert your target and models to use of the **Hardware Implementation** pane. See “Configure Production and Test Hardware” (Simulink Coder).

Customize Build to Use Shared Utility Code

The shared utility folders (`slprj/`*`target`*`/_sharedutils`) typically store generated utility code that is common to a top model and the models it references. You can also force the build process to use a shared utilities folder for a standalone model. See "Specify Generated Code Interfaces" (Simulink Coder) for details.

If you want your target to support compilation of code generated in the shared utilities folder, you must modify your template makefile (TMF). The shared utilities folder is required to support model reference builds. See "Support Model Referencing" on page 71-83 to learn about additional updates for supporting model reference builds.

The exact syntax of the changes can vary due to differences in the `make` utility and compiler/archive tools used by your target. The examples below are based on the Free Software Foundation's GNU make utility. You can find the following updated TMF examples for GNU and Microsoft Visual C++ make utilities in the GRT and ERT target folders:

- GRT: *`matlabroot`*`/rtw/c/grt` (open)
  - `grt_lcc.tmf`
  - `grt_vc.tmf`
  - `grt_unix.tmf`
- ERT: *`matlabroot`*`/rtw/c/ert` (open)
  - `ert_lcc.tmf`
  - `ert_vc.tmf`
  - `ert_unix.tmf`

Use the GRT or ERT examples as a guide to the location, within the TMF, of the changes and additions described below.

Note The ERT-based TMFs contain extra code to handle generation of ERT S-functions and model reference simulation targets. Your target does not need to handle these cases.

Modify Template Makefiles to Support Shared Utilities

Make the following changes to your TMF to support the shared utilities folder:

1. Add the following make variables and tokens to be expanded when the makefile is generated:

```
SHARED_SRC      = |>SHARED_SRC<|
SHARED_SRC_DIR  = |>SHARED_SRC_DIR<|
SHARED_BIN_DIR  = |>SHARED_BIN_DIR<|
SHARED_LIB      = |>SHARED_LIB<|
```

   SHARED_SRC specifies the shared utilities folder location and the source files in it. A typical expansion in a makefile is

```
SHARED_SRC      = ../slprj/ert/_sharedutils/*.c
```

   SHARED_LIB specifies the library file built from the shared source files, as in the following expansion.

```
SHARED_LIB      = ../slprj/ert/_sharedutils/rtwshared.lib
```

   SHARED_SRC_DIR and SHARED_BIN_DIR allow specification of separate folders for shared source files and the library compiled from the source files. In the current release, TMFs use the same path, as in the following expansions.

```
SHARED_SRC_DIR  = ../slprj/ert/_sharedutils
SHARED_BIN_DIR  = ../slprj/ert/_sharedutils
```

2. Set the SHARED_INCLUDES variable according to whether shared utilities are in use. Then append it to the overall INCLUDES variable.

```
SHARED_INCLUDES =
ifneq ($(SHARED_SRC_DIR),)
SHARED_INCLUDES = -I$(SHARED_SRC_DIR)
endif

INCLUDES = -I. $(MATLAB_INCLUDES) $(ADD_INCLUDES) \
           $(USER_INCLUDES) $(SHARED_INCLUDES)
```

3. Update the SHARED_SRC variable to list shared files explicitly.

```
SHARED_SRC := $(wildcard $(SHARED_SRC))
```

4. Create a SHARED_OBJS variable based on SHARED_SRC.

```
SHARED_OBJS = $(addsuffix .o, $(basename $(SHARED_SRC)))
```

5. Create an OPTS (options) variable for compilation of shared utilities.

```
SHARED_OUTPUT_OPTS = -o $@
```

6. Provide a rule to compile the shared utility source files.

```
$(SHARED_OBJS) : $(SHARED_BIN_DIR)/%.o : $(SHARED_SRC_DIR)/%.c
 $(CC) -c $(CFLAGS) $(SHARED_OUTPUT_OPTS) $<
```

7 Provide a rule to create a library of the shared utilities. The following example is based on The Open Group UNIX platforms.

```
$(SHARED_LIB) : $(SHARED_OBJS)
   @echo "### Creating $@ "
   ar r $@ $(SHARED_OBJS)
   @echo "### Created  $@ "
```

8 Add `SHARED_LIB` to the rule that creates the final executable.

```
$(PROGRAM) : $(OBJS) $(LIBS) $(SHARED_LIB)
$(LD) $(LDFLAGS) -o $@ $(LINK_OBJS) $(LIBS) $(SHARED_LIB)\
    $(SYSLIBS)
@echo "### Created executable: $(MODEL)"
```

9 Remove explicit reference to `rt_nonfinite.c` or `rt_nonfinite.cpp` from your TMF. For example, change

```
ADD_SRCS = $(RTWLOG) rt_nonfinite.c
```

to

```
ADD_SRCS = $(RTWLOG)
```

Custom Target Development in Simulink Coder

About Embedded Target Development

Target files bundled with the code generator are suitable for many different applications and development environments. Third-party targets provide additional versatility. In addition, you have the option of implementing a custom target.

To implement a target based on the ARM Cortex®-A or ARM Cortex-M processor, install the corresponding support package and see the Target SDK: Embedded Coder Support Package for ARM Cortex-A Processors, "Develop a Target" (Embedded Coder Support Package for ARM Cortex-A Processors) or Embedded Coder Support Package for ARM Cortex-M Processors, "Develop a Target" (Embedded Coder Support Package for ARM Cortex-M Processors). Otherwise, use these functions and topics.

Custom Targets

You might want to implement a custom target for one of the following reasons:

- To enable end users to generate executable production code for a specific CPU or development board, using a specific development environment (compiler/linker/debugger).
- To support I/O devices on the target hardware by incorporating custom device driver blocks into your models.
- To configure the build process for a special compiler (such as a cross-compiler for an embedded microcontroller or DSP board) or development/debugging environment.

The code generator provides a point of departure for the creation of custom embedded targets, for the basic purposes above. This manual covers the tasks and techniques you need to implement a custom embedded target.

Types of Targets

The following sections describe several types of targets intended for different use cases

About Target Types

There is a progression of capabilities from baseline or rapid prototyping targets to production targets. Initially, you might want to implement a rapid prototyping target. Later, you can enhance the target to be more full-featured. For example, you might want to add support for software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation at some point for verifying your embedded target. The target types are not mutually exclusive. An embedded target can support more than one of these use cases, or additional uses not outlined here.

The discussion of target types is followed by "Recommended Features for Embedded Targets" on page 71-4, which contains a suggested list of features and general guidelines for embedded target development.

Rapid Prototyping Targets

A *rapid prototyping target* or baseline target offers a starting point for targeting a production processor. A rapid prototyping target integrates coded generator software with one or more popular cross-development environments (compiler/linker/debugger tool chains). A rapid prototyping target provides a starting point from which you can customize the target for application needs.

Target files provided for this type of target should be readable, easy to understand, and fully commented and documented. Specific attention should be paid to the interface to the intended cross-development environment. This interface should be implemented using the preferred approach for that particular development system. For example, some development environments use traditional make utilities, while others are based on project-file builds that can be automated under control of the code generator.

When you use a rapid prototyping target, you need to include your own device driver and legacy code and modify linker memory maps to suit your needs. You should be familiar with the targeted development system.

Production Targets

A *production target* supports embedded application development for a production processor. It includes the capability to create program executables that interact immediately with the external world. In general, ease of use is more important than simplicity or readability of generated code files, because it is assumed that you do not want or need to modify the files.

Desirable features for a production target include:

- Significant I/O driver support, provided out of the box
- Easy downloading of generated standalone executable programs with third-party debuggers
- User-controlled placement of an executable in FLASH or RAM memory
- Support for code visibility and tuning on target hardware

Verifying Targets With SIL and PIL Simulation

You can use software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation to verify generated code and validate the target compiler/processor environment.

You can use SIL and PIL simulation modes to verify automatically generated code by comparing the results with a normal mode simulation. With SIL, you can easily verify the behavior of production-intent source code on your host computer; however, it is generally not possible to verify exactly the same code that will subsequently be compiled for your target hardware because the code must be compiled for your host platform (i.e. a different compiler and different processor architecture than the target). With PIL simulation, you can verify exactly the same code that you intend to deploy in production, and you can run the code either on real target hardware or on an instruction set simulator.

For examples describing how to run processor-in-the-loop testing to verify a custom target, see “Sample Custom Targets” on page 71-9.

For more information on SIL and PIL simulation, see “SIL and PIL Simulations” on page 64-2.

HIL Simulation

A specialized use case is the generation of executables intended for use in *hardware-in-the-loop* (HIL) simulations. In a HIL simulation, parts of a pure simulation are gradually replaced with hardware components as components are refined and fabricated. HIL simulation offers an efficient design process that eliminates costly iterations of part fabrication.

Recommended Features for Embedded Targets

- "Deployment and Hardware Issues" on page 71-7

Basic Target Features

- You can base targets on the generic real-time (GRT) target or the Embedded Real-Time (ERT) target that is included in the Embedded Coder product.

  If your target is based on the ERT target, it should use `'Embedded-C'` value for the `CodeFormat` TLC variable, and it should inherit the options defined in the ERT target's system target file with the following lines in the TLC file:

  ```
  %% Assign code format
  %assign CodeFormat = "Embedded-C"
  %%-----------------------------
  /%
    BEGIN_RTW_OPTIONS
    rtwgensettings.DerivedFrom = 'ert.tlc';
    END_RTW_OPTIONS
  %/
  %%-----------------------------
  ```

  By following these recommendations, your target has the production code generation capabilities of the ERT target.

  See "Customize System Target Files" on page 71-29 for further details on the inheritance mechanism, setting the `CodeFormat`, and other details.

- The most fundamental requirement for an embedded target is that it generate a real-time executable from a model or subsystem. Typically, an embedded target generates a timer interrupt-based, bareboard executable (although targets can be developed for an operating system environment as well).

  Your target should support code generator concepts of single-tasking and multitasking solver modes for model execution. Tasking support is available with the ERT target, but you should thoroughly understand how it works before implementing an ERT-based target.

  For information on timer interrupt-based execution, see "Absolute and Elapsed Time Computation" (Simulink Coder) and "Asynchronous Events" (Simulink Coder).

- You should generate the target executable's main program module, rather than using a static main module (such as the static `rt_main.c` or `rt_cppclass_main.cpp` module provided with the software). A generated `main.c` or `.cpp` can be made much more readable and more efficient, since it omits preprocessor checks and other extra code.

For information on generated and static main program modules, see “Deploy Generated Standalone Executable Programs To Target Hardware” on page 49-2.

- Follow the guidelines in “Folder and File Naming Conventions” on page 71-11.

Integration with Target Development Environments

- Most cross-development systems run under a Microsoft Windows PC host. Your target should support the Windows operating system as the host environment.

  Some cross-development systems support one or more versions of The Open Group UNIX platforms, allowing for UNIX host support as well.
- Your embedded target must support at least one embedded development environment. The interface to a development environment can take one of several forms. The toolchain approach and template makefile approach generate standard makefiles to work with your development environment. For general information about these build approaches, see “Choose and Configure Build Process” (Simulink Coder). For detailed information about the structure of template makefiles, see “Customize Template Makefiles” on page 71-62.

  Another approach with IDE-based tools is to create a Microsoft Visual Studio Solution from your target for integration within a Visual Studio project.

  It is important to consider the license requirements and restrictions of the development environment vendor. You may need to modify files provided by the vendor and ship them as part of the embedded target.

  See “Interface to Development Tools” on page 71-104 for further information.

Observing Execution of Target Code

- Your target should support a mechanism you can use to observe the target code as it runs in real time (outside of a debugger).

  You can use the `rtiostream` API to implement a communication channel to enable exchange of data between different processes. For an example of creating a communication channel for target connectivity, see “Create a Target Communication Channel for Processor-In-The-Loop (PIL) Simulation”. This `rtiostream` communication channel is required to enable processor-in-the-loop (PIL) on a new target. See “Communications `rtiostream` API” on page 64-46.

One industry-standard approach is to use the CAN bus, with an ASAP2 file and CAN Calibration Protocol (CCP). There are several host-based graphical front-end tools available that connect to a CCP-enabled target and provide data viewing and parameter tuning. Supporting these tools requires implementation of CAN hardware drivers and CCP protocol for the target, as well as ASAP2 file generation. Your target can leverage the ASAP2 support provided with the code generator.

Another option is to support Simulink external mode over a serial interface (RS-232). See the "What You Can Do with a Host/Target Communication Channel" (Simulink Coder) for information on using the external mode API.

Deployment and Hardware Issues

- Device driver support is an important issue in the design of an embedded target. Device drivers are Simulink blocks that support either hardware I/O capabilities of the target CPU, or I/O features of the development board.

  If you are developing a rapid prototyping target, consider providing minimal driver support, on the assumption that end users develop their own drivers. If you are developing a production target, you should provide full driver support.

  See "Device Drivers" (Simulink Coder).

- Automatic download of generated code to the target hardware makes a target easier to use. Typically a debugger utility is used; if the chosen debugger supports command script files, this can be straightforward to implement. "STF_make_rtw_hook.m" on page 71-21 describes a mechanism to execute code from the build process. You can use this mechanism to make `system()` calls to invoke utilities such as a debugger. You can invoke other simple downloading utilities in a similar fashion.

  If your development system supports COM automation, you can control the download process by that mechanism. Using COM automation is discussed in "Interface to Development Tools" on page 71-104.

- Executables that are mapped to RAM memory are typical. You can provide optional support for FLASH or RAM placement of the executable by using your target's code generation options. To support this capability, you might need multiple linker command files, multiple debugger scripts, and possibly multiple makefiles or project files. Also include the ability to automatically switch between these files, depending on the RAM/FLASH option value.

- Select a popular, widely available evaluation or prototype board for your target processor. Consider enclosed and ruggedized versions of the target board. Also consider board level support for the various on-chip I/O capabilities of the target CPU, and the availability of development systems that support the selected board.

More About

- "Sample Custom Targets" (Simulink Coder)
- "Target Development Folders, Files, and Builds" (Simulink Coder)
- "Customize System Target Files" (Simulink Coder)
- "Customize Template Makefiles" (Simulink Coder)
- "Custom Target Optional Features" (Simulink Coder)
- "Support Toolchain Approach with Custom Target" (Simulink Coder)
- "Support Model Referencing" (Simulink Coder)
- "Support Compiler Optimization Level Control" (Simulink Coder)
- "Support C Function Prototype Control" (Simulink Coder)
- "Support C++ Class Interface Control" (Simulink Coder)
- "Support Concurrent Execution of Multiple Tasks" (Simulink Coder)
- "Interface to Development Tools" (Simulink Coder)
- "Device Drivers" (Simulink Coder)

Sample Custom Targets

There are technical solutions on the MathWorks Web site that you can use as a starting point to create your own target solution. The solutions provide guides to the following tasks for creating custom targets:

- Methods of embedding code onto a custom processor
- Creating a system target file
- Customizing the makefile and main file
- Adding compiler, chip, and board specific information
- Integrating legacy code and device drivers
- Creating blocks and libraries
- Implementing processor-in-the-loop (PIL) testing.

1. Start by downloading the embedded targets development guide `zip` file from this web page:

   Is there an example guide on developing an embedded target...?

   The `zip` file provides example files and a guide to developing a custom embedded target. The guide is divided into two parts, one on creating a generic custom target and another on creating a target for the Freescale™ S12X processor using the Cosmic Compiler.

   Read the example guide along with this document to understand the tasks for developing embedded targets.

2. For more detailed example files for specific processors, see:

   - Is there an example Freescale S12X target... using the Cosmic Compiler?
   - Is there an example Freescale S12X target... using the CodeWarrior Compiler?

   These example kits contain example models, code generation files, and instruction guides on generating and testing code for the processor. The Cosmic example illustrates the use of the target connectivity API for processor-in-the-loop (PIL) testing. The CodeWarrior example does not have PIL but shows CAN Calibration Protocol (CCP) and Simulink external mode.

   The intent of the example kits is to provide working examples that you can use as a base to create your own target solution. The intent is not to provide a full featured

and maintained Embedded Target product like those provided by MathWorks or third-party products, as listed on the Embedded Coder Hardware Support Web page.

3 You can watch videos showing overviews of both the example kits at the following links:

- `www.mathworks.com/videos/programming-the-freescale-s12x-target-68811.html`
- `www.mathworks.com/videos/programming-arm9-using-the-hitex-str9-comstick-68812.html`

For another example target for the ARM9 (STR9) processor, see Is there an example ARM9 (STR9) target... using the GNU ARM Compiler and Hitex STR9-comStick?.

If you have questions on specific targets, please email mytarget@mathworks.com.

The example kits and this document describe Embedded Coder features such as customized `ert` system target files and processor-in-the-loop testing, but you can study the examples as a starting point for use with Simulink Coder targets.

More About

- "About Embedded Target Development" (Simulink Coder)
- "Customize System Target Files" (Simulink Coder)
- "Customize Template Makefiles" (Simulink Coder)
- "Custom Target Optional Features" (Simulink Coder)
- "Support Toolchain Approach with Custom Target" (Simulink Coder)
- "Interface to Development Tools" (Simulink Coder)
- "Device Drivers" (Simulink Coder)

Target Development Folders, Files, and Builds

Target development mechanics work with a number of folder and file types. The following topics provide the information to develop custom targets, configure folder usage, and use custom targets in the build process.

In this section...

Folder and File Naming Conventions

You can use a single folder for your custom target files, or if desired you can use subfolders, for example containing files associated with specific development environments or tools.

For a custom target implementation, the recommended folder and file naming conventions are

- Use *only* lowercase in folder names, filenames, and extensions.
- Do not embed spaces in folder names. Spaces in folder names cause errors with many third-party development environments.
- Include desired folders in the MATLAB path
- Do *not* place your custom target folder anywhere in the MATLAB folder tree (that is, in or under the `matlabroot` folder). If you place your folder under `matlabroot` you risk losing your work if you install a new MATLAB version (or reinstall the current version).

The following sections explain how to organize your target folders and files and add them to the your MATLAB path. They also provide high-level descriptions of the files.

In this document, `mytarget` is a placeholder name that represents folders and files that use the target's name. The names `dev_tool1`, `dev_tool2`, and so on represent

subfolders containing files associated with development environments or tools. This document describes an example structure where the folder `mytarget` contains subfolders for `mytarget`, `blocks`, `dev_tool1`, `dev_tool2`. The top level folder `mytarget` is the *target root folder*.

Components of a Custom Target

Overview

The components of a custom target are files located in a hierarchy of folders. The top-level folder in this structure is called the *target root folder*. The target root folder and its contents are named, organized, and located on the MATLAB path according to conventions described in "Folder and File Naming Conventions" on page 71-11.

The components of a custom target include

- Code components: C source code that supervises and supports execution of generated model code.
- Control files:
  - A system target file (STF) to control the code generation process.
  - File(s) to control the building of an executable from the generated code. In a traditional make-based environment, a template makefile (TMF) generates a makefile for this purpose. Another approach is to generate project files in support of a modern integrated development environment (IDE) such as the Freescale Semiconductor CodeWarrior IDE.
  - Hook files: Optional TLC and MATLAB program files that can be invoked at well-defined stages of the build process. Hook files let you customize the build process and communicate information between various phases of the process.
- Other target files: Files that let you integrate your target into the MATLAB environment. For example, you can provide an `info.xml` file to make your target block libraries and examples available from a MATLAB session.

The next sections introduce key concepts and terminology you need to know to develop each component. References to more detailed information sources are provided.

Code Components

An executable program containing code generated from a Simulink model consists of a number of code modules and data structures. These fall into two categories.

Application Components

Application components are those which are specific to a particular model; they implement the functions represented by the blocks in the model. Application components are not specific to the target. Application components include

- Modules generated from the model
- User-written blocks (S-functions)
- Parameters of the model that are visible, and can be interfaced to, external code

Execution Support Files

A number of code modules and data structures, referred to collectively as the *execution support files*, are responsible for managing and supporting the execution of the generated program. The execution support files modules are not automatically generated. Depending on the requirements of your target, you must implement certain parts of the execution support files. Execution Support Files summarizes the execution support files.

Execution Support Files

You Provide...	The Code Generator Provides...
Customized main program	Generic main program
Timer interrupt handler to run model	Execution engine and integration solver (called by timer interrupt handler)
Other interrupt handlers	Example interrupt handlers (Asynchronous Interrupt blocks)
Device drivers	Example device drivers
Data logging, parameter tuning, signal monitoring, and external mode support	Data logging, parameter tuning, signal monitoring, and external mode APIs

User-Written Execution Support Files

The code generator provides most of the execution support files. Depending on the requirements of your target, you must implement some or all of the following elements:

- A timer *interrupt service routine* (ISR). The timer runs at the program's base sample rate. The timer ISR is responsible for operations that must be completed within a single clock period, such as computing the current output sample. The timer ISR usually calls the `rt_OneStep` function.

  If you are targeting a real-time operating system (RTOS), your generated code usually executes under control of the timing and task management mechanisms provided by the RTOS. In this case, you may not have to implement a timer ISR.

- The *main program*. Your main program initializes the blocks in the model, installs the timer ISR, and executes a background task or loop. The timer periodically interrupts the main loop. If the main program is designed to run for a finite amount of time, it is also responsible for cleanup operations — such as memory deallocation and masking the timer interrupt — before terminating the program.

  If you are targeting a real-time operating system (RTOS), your main program most likely spawns tasks (corresponding to the sample rates used in the model) whose execution is timed and controlled by the RTOS.

  Your main program typically is based on a generated or static main program. For details on the structure of the execution support files, code execution, and guidelines for customizing main programs, see “Deploy Generated Standalone Executable Programs To Target Hardware” on page 49-2.

- *Device drivers*. Drivers communicate with I/O devices on your target hardware. In production code, device drivers are normally implemented as inlined S-functions.

- *Other interrupt handlers*. If your models need to support asynchronous events, such as hardware generated interrupts and asynchronous read and write operations, you must supply interrupt handlers. The Interrupt Templates library provides examples.

- *Data logging, parameter tuning, signal monitoring, and external mode support*. It is atypical to implement rapid prototyping features such as external mode support in an embedded target. However, it is possible to support these features by using standard code generator APIs. See “Data Exchange Interfaces” (Simulink Coder) for details.

Control Files

The code generation and build process is directed by a number of TLC and MATLAB files collectively called *control files*. This section introduces and summarizes the main control files.

Top-Level Control File (make_rtw)

The build process is initiated when you press **Ctrl+B**. At this point, the build process parses the **Make command** field of the **Code Generation** pane of the Configuration Parameters dialog box, expecting to find the name of a MATLAB command that controls the build process (as well as optional arguments to that command). The default command is `make_rtw`, and the default top-level control file for the build process is `make_rtw.m`.

Note: `make_rtw` is an internal MATLAB command used by the build process. Normally, target developers do not need detailed knowledge of how `make_rtw` works. (The details for target developers are described in "Target Development and the Build Process" on page 71-23.) You should not invoke `make_rtw` directly from MATLAB code, and you should not customize `make_rtw.m`.

The `make_rtw.m` file contains the logic required to execute your target-specific control files, including a number of hook points for execution of your custom code. `make_rtw` does the following:

- Passes optional arguments in to the build process
- Performs required preprocessing before code generation
- Executes the STF to perform code generation (and optional HTML report generation)
- Processes the TMF to generate a makefile
- Invokes a make utility to execute the makefile and build an executable
- Performs required post-processing (such as generating calibration data files or downloading the generated executable to the target)

System Target File (STF)

The Target Language Compiler (TLC) generates target-specific C or C++ code from an partial description of your Simulink block diagram (*`model`*`.rtw`). The Target Language Compiler reads *`model`*`.rtw` and executes a program consisting of several target files (`.tlc` files.) The STF, at the top level of this program, controls the code generation process. The output of this process is a number of source files, which are fed to your development system's make utility.

You need to create a customized STF to set code generation parameters for your target. You should copy, rename, and modify the standard ERT system target file (*`matlabroot`*`/rtw/c/ert/ert.tlc`).

The detailed structure of the STF is described in "Customize System Target Files" on page 71-29.

Note: The STF selects whether the target supports the toolchain approach or template makefile approach for code generation. See "Customize System Target Files" on page 71-29.

Template Makefile (TMF)

A TMF provides information about your model and your development system. The build process uses this information to create a makefile (`.mk` file) that builds an executable program.

Some targets implement more than one TMF, in order to support multiple development environments (for example, two or more cross-compilers) or multiple modes of code generation (for example, generating a binary executable versus generating a project file for your compiler).

The Embedded Coder software provides a large number of TMFs suitable for different types of development computer systems. These TMFs are located in *`matlabroot`*`/rtw/c/ert` (open). The standard TMFs are described in "Template Makefiles and Make Options" (Simulink Coder).

The detailed structure of the TMF is described in "Customize Template Makefiles" on page 71-62.

Note: The STF selects whether the target supports the toolchain approach or template makefile approach for code generation. See "Customize System Target Files" on page 71-29.

Hook Files

The build process allows you to 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.

The hook files must follow well-defined naming and location requirements. "Folder and File Naming Conventions" on page 71-11 describes these requirements.

Key Folders Under Target Root (mytarget)

Target Root Folder (mytarget)

This folder contains the key subfolders for the target (see "Folder and File Naming Conventions" on page 71-11). You can also locate miscellaneous files (such as a `readme` file) in the target root folder. The following sections describe required and optional subfolders and their contents.

Target Folder (mytarget/mytarget)

This folder contains files that are central to the target, such as the system target file (STF) and template makefile (TMF). "Key Files in Target Folder (mytarget/mytarget)" on page 71-19 summarizes the files that should be stored in `mytarget/mytarget`, and provides pointers to detailed information about these files.

Note `mytarget/mytarget` should be on the MATLAB path.

Target Block Folder (mytarget/blocks)

If your target includes device drivers or other blocks, locate the block implementation files in this folder. `mytarget/blocks` contains

- Compiled block MEX-files
- Source code for the blocks
- TLC inlining files for the blocks
- Library models for the blocks (if you provide your blocks in one or more libraries)

Note `mytarget/blocks` should be on the MATLAB path.

You can also store example models and supporting files in `mytarget/blocks`. Alternatively, you can create a `mytarget/mytargetdemos` folder, which should also be on the MATLAB path.

To display your blocks in the standard Simulink Library Browser and/or integrate your example models into the MATLAB session environment , you can create the files described below and store them in `mytarget/blocks`.

mytarget/blocks/slblocks.m

This file allows a group of blocks to be integrated into the Simulink Library and Simulink Library Browser.

Example slblocks.m File

```
function blkStruct = slblocks
% Information for "Blocksets and Toolboxes" subsystem
blkStruct.Name = sprintf('Embedded Target\n for MYTARGET');
blkStruct.OpenFcn = 'mytargetlib';
blkStruct.MaskDisplay = 'disp(''MYTARGET'')';

% Information for Simulink Library Browser
Browser(1).Library = 'mytargetlib';
Browser(1).Name    = 'Embedded Target for MYTARGET';
Browser(1).IsFlat  = 1;% Is this library "flat" (i.e. no subsystems)?

blkStruct.Browser = Browser;
```

mytarget/blocks/demos.xml

This file provides information about the components, organization, and location of example models. MATLAB software uses this information to place the example in the MATLAB session environment.

Example demos.xml File

```
<?xml version="1.0" encoding="utf-8"?>
<demos>
   <name>Embedded Target for MYTARGET</name>
   <type>simulink</type>
   <icon>$toolbox/matlab/icons/boardicon.gif</icon>
   <description source = "file">mytarget_overview.html</description>

   <demosection>
      <label>Multirate model</label>
      <demoitem>
         <label>MYTARGET demo</label>
         <file>mytarget_overview.html</file>
         <callback>mytarget_model</callback>
```

```
    </demoitem>
  </demosection>

</demos>
```

Development Tools Folder (mytarget/dev_tool1, mytarget/dev_tool2)

These folders contain files associated with specific development environments or tools (`dev_tool1`, `dev_tool2`, etc.). Normally, your target supports at least one such development environment and invokes its compiler, linker, and other utilities during the build process. `mytarget/dev_tool1` includes linker command files, startup code, hook functions, and other files required to support this process.

For each development environment, you should provide a separate folder.

Target Source Code Folder (mytarget/src)

This folder is optional. If the complexity of your target requires it, you can use `mytarget/src` to store common source code and configuration code (such as boot and startup code).

Key Files in Target Folder (mytarget/mytarget)

Introduction

The target folder `mytarget/mytarget` contains key files in your target implementation. These include the system target file, template makefile, main program module, and optional M and TLC hook files that let you add target-specific actions to the build process. The following sections describe the key target folder files.

mytarget.tlc

`mytarget.tlc` is the system target file (STF). Functions of the STF include

- Making the target visible in the System Target File Browser
- Definition of code generation options for the target (inherited and target-specific)
- Providing an entry point for the top-level control of the TLC code generation process

You should base your STF on `ert.tlc`, the STF provided by Embedded Coder software.

"Customize System Target Files" on page 71-29 gives detailed information on the structure of the STF, and also gives instructions on how to customize an STF to

- Display your target in the System Target File Browser
- Add your own target options to the Configuration Parameters dialog box
- Tailor the code generation and build process to the requirements of your target

mytarget.tmf

`mytarget.tmf` is the template makefile for building an executable for your target.

For basic information on the structure and operation of template makefiles, see "Customize Template Makefiles" on page 71-62.

If your target development environment requires automation of a modern integrated development environment (IDE) rather than use of a traditional make utility, see "Interface to Development Tools" on page 71-104.

mytarget_genfiles.tlc

This file is optional. `mytarget_genfiles.tlc` is useful as a central file from which to invoke target-specific TLC files that generate additional files as part of your target build process. For example, your target may create sub-makefiles or project files for a development environment, or command scripts for a debugger to do automatic downloads. See "Using mytarget_genfiles.tlc" on page 71-45 for details.

mytarget_main.c

A main program module is required for your target. To provide a main module, you can either

- Modify the `rt_main.c` or `rt_cppclass_main.cpp` module provided by the software

- Generate `mytarget_main.c` or `.cpp` during the build process

For a description of the operation of main programs, see "Deploy Generated Standalone Executable Programs To Target Hardware" on page 49-2. The section also contains guidelines for generating and modifying a main program module.

STF_make_rtw_hook.m

STF`_make_rtw_hook.m` is an optional hook file that you can use to invoke target-specific functions or executables at specified points in the build process. *STF*`_make_rtw_hook.m` implements a function that dispatches to a specific action depending on the `method` argument that is passed into it.

"Customize Build Process with STF_make_rtw_hook File" (Simulink Coder) describes the operation of the *STF*`_make_rtw_hook.m` hook file in detail.

STF_rtw_info_hook.m (obsolete)

Prior to 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 the properties that were formerly specified in your *STF*`_rtw_info_hook` file.

For backward compatibility, existing *STF*`_rtw_info_hook` files are still available. However, you should convert your target and models to use the **Hardware Implementation** pane. See "Configure Run-Time Environment Options" (Simulink Coder).

info.xml

This file provides information to MATLAB software that specifies where to display the target toolbox in the MATLAB session environment. For more information, see "Display Custom Documentation" (MATLAB).

mytarget_overview.html

By convention, this file serves as home page for the target examples.

The <description> field in demos.xml should point to mytarget_overview.html (see “mytarget/blocks/demos.xml” on page 71-18).

Example mytarget_overview.html File

```
<html>
<head><title>Embedded Target for MYTARGET</title></head><body>
<p style="color:#990000; font-weight:bold; font-size:x-large">Embedded Target
for MYTARGET Example Model</p>

<p>This example provides a simple model that allows you to generate an executable
for a supported target board. You can then download and run the executable and
set breakpoints to study and monitor the execution behavior.</p>

</body>
</html>
```

Additional Files for Externally Developed Targets

Introduction

If you are developing an embedded target that is not installed into the MATLAB tree, you should provide a target setup script and target documentation within mytarget/mytarget, for the convenience of your users. The following sections describe the required materials and where to place them.

mytarget/mytarget/mytarget_setup.m

This file script adds paths for your target to the MATLAB path. Your documentation should instruct users to run the script when installing the target.

You should include a call to the MATLAB function savepath in your mytarget_setup.m script. This function saves the added paths, so users need to run mytarget_setup.m only once.

The following code is an example mytarget_setup.m file.

```
function mytarget_setup()
curpath = pwd;
tgtpath = curpath(1:end-length('\mytarget'));
```

```
addpath(fullfile(tgtpath, 'mytarget'));
addpath(fullfile(tgtpath, 'dev_tool1'));
addpath(fullfile(tgtpath, 'blocks'));
addpath(fullfile(tgtpath, 'mytargetdemos'));
savepath;
disp('MYTARGET Target Path Setup Complete.');
```

mytarget/mytarget/doc

You should put the documentation related to your target in the folder `mytarget/mytarget/doc`.

Target Development and the Build Process

About the Build Process

To develop an embedded target, you need a thorough understanding of the build process. Your embedded target uses the build process and may require you to modify or customize the process. A general overview of code generation and the build process is given in "Code Generation" (Simulink Coder) and "Build Process" (Simulink Coder).

This section supplements that overview with a description of the build process as customized by the Embedded Coder software. The emphasis is on points in the process where customization hooks are available and on passing information between different phases of the process.

This section concludes with "Additional Information Passing Techniques" on page 71-26, describing assorted tips and tricks for passing information during the build process.

Build Process Phases and Information Passing

It is important to understand where (and when) the build process obtains required information. Sources of information include

- The *`model`*`.rtw` file, which provides information about the generating model. The information in *`model`*`.rtw` is available to target TLC files.

- The code generation panes of the Configuration Parameters dialog box. Options (both general and target-specific) are provided through check boxes, menus, and edit fields. You can associate options with TLC variables in the `rtwoptions` data structure. Use the **Configuration Parameters** > **Code Generation** > **Custom Code** > **Additional build information** > **Defines** field to define makefile tokens .
- The selected toolchain (for toolchain approach builds) or selected template makefile `.tmf` (for template makefile approach builds); these generate the model-specific makefile.
- Environment variables on the host computer. Environment variables provide additional information about installed development tools.
- Other target-specific files such as target-related TLC files, linker command files, or project files.

It is also important to understand the several phases of the build process and how to pass information between the phases. The build process comprises several high-level phases:

- Execution of the top-level file (`slbuild.m` or `rtwbuild.m`) to sequence through the build process for a target
- Conversion of the model into the TLC input file (*`model`*`.rtw`)
- Generation of the target code by the TLC compiler
- Compilation of the generated code with `make` or other utilities
- Transmission of the final generated executable to the target hardware with a debugger or download utility

It is helpful to think of each phase of the process as a different “environment” that maintains its own data. These environments include

- MATLAB code execution environment (MATLAB)
- Simulink
- Target Language Compiler execution environment
- makefile
- Development environments such as and IDE or debugger

In each environment, you might get information from the various sources mentioned above. For example, during the TLC phase, execute MATLAB file might execute to obtain information from the MATLAB environment. Also, a given phase may generate information for a subsequent phase.

See “Key Files in Target Folder (mytarget/mytarget)” on page 71-19 for details on the available MATLAB file and TLC hooks for information passing, with code examples.

Additional Information Passing Techniques

This section describes a number of useful techniques for passing information among different phases of the build process.

tlcvariable Field in rtwoptions Structure

Parameters on the code generation panes of the Configuration Parameters dialog box can be associated with a TLC variable, and specified in the `tlcvariable` field of the option's entry in the `rtwoptions` structure. The variable value is passed on the command line when TLC is invoked. This provides a way to make code generation parameters and their values available in the TLC phase.

See “System Target File Structure” on page 71-30 for further information.

makevariable Field in rtwoptions Structure

You can associate code generation parameters with a template makefile token, that you specify in the `makevariable` field of the option's entry in the `rtwoptions` structure. If a token of the same name as the `makevariable` name exists in the TMF, the token is updated with the option value when the final makefile is created. If the token does not exist in the TMF, the `makevariable` is passed in on the command line when make is invoked. Thus, in either case, the `makevariable` is available to the makefile.

See “System Target File Structure” on page 71-30 for further information.

Accessing Host Environment Variables

You can access host shell environment variables at the MATLAB command line by entering the `getenv` command. For example:

```
getenv ('MSDEVDIR')

ans =

D:\Applications\Microsoft Visual Studio\Common\MSDev98
```

To access the same information from TLC, use the `FEVAL` directive to invoke `getenv`.

```
%assign eVar = FEVAL("getenv","<varname>")
```

Supplying Development Environment Information to Your Template Makefile

An embedded target must tie the build process to target-specific development tools installed on a host computer. For the make process to run these tools, the TMF must

be able to determine the name of the tools, the path to the compiler, linker, and other utilities, and possibly the host operating system environment variable settings.

Require the end user to modify the target TMF. The user enters path information (such as the location of a compiler executable), and possibly host operating system environment variables, as make variables. This allows the TMF to be tailored to specific needs.

Using MATLAB Application Data

Application data provides a way for applications to save and retrieve data stored with the GUI. This technique enables you to create what is essentially a user-defined property for an object, and use this property to store data for use in the build process. If you are unfamiliar with this technique for creating graphical user interfaces, see "Store Data as Application Data" (MATLAB).

The following code examples illustrates the use of application data to pass information to TLC.

This file, `tlc2appdata.m`, stores the `data` passed in as application data under the name passed in (`appDataName`).

```
function k = tlc2appdata(appDataName,data)
  disp([mfilename,': ',appDataName,' ', data]);
  setappdata(0,appDataName,data);
  k = 0;  % TLC expects a return value for FEVAL.
```

The following sample TLC file uses the `FEVAL` directive to invoke `tlc2appdata.m` to store arbitrary application data, under the name `z80`.

```
%% test.tlc
%%
%assign myApp = "z80"
%assign myData = "314159"
%assign dummy = FEVAL("tlc2appdata",myApp,myData)
```

To test this technique:

1. Create the `tlc2appdata.m` file as shown. Check that `tlc2appdata.m` is stored in a folder on the MATLAB path.
2. Create the TLC file as shown. Save it as `test.tlc`.
3. Enter the following command at the MATLAB prompt to execute the TLC file:

   ```
   tlc test.tlc
   ```

4 Get the application data at the MATLAB prompt:

```
k = getappdata(0,'z80')
```

The function returns the value 314159.

5 Enter the following command.

```
who
```

Note that application data is not stored in the MATLAB workspace. Also observe that the z80 data is not visible. Using application data in this way has the advantage that it does not clutter the MATLAB workspace. Also, it helps prevent you from accidently deleting your data, since it is not stored directly in the your workspace.

A real-world use of application data might be to collect information from the *`model`*`.rtw` file and store it for use later in the build process.

Adding Block-Specific Information to the Makefile

The `rtwmakecfg` mechanism provides a method for inlined S-functions such as driver blocks to add information to the makefile. This mechanism is described in "Use rtwmakecfg.m API to Customize Generated Makefiles" (Simulink Coder).

More About

- "About Embedded Target Development" (Simulink Coder)

Customize System Target Files

This section provides information on the structure of the STF, guidelines for customizing an STF, and a basic tutorial that helps you get a skeletal STF up and running.

In this section...

Control Code Generation With the System Target File

The system target file (STF) exerts overall control of the code generation stage of the build process. The STF also lets you control the presentation of your target to the end user. The STF provides

- Definitions of variables that are fundamental to the build process, such as the value for the `CodeFormat` TLC variable
- The main entry point to the top-level TLC program that generates code
- Target information for display in the System Target File Browser
- A mechanism for defining target-specific code generation options (and other parameters related to the build process) and for displaying them in the Configuration Parameters dialog box
- A mechanism for inheriting options from another target (such as the Embedded Real-Time (ERT) target)

Note that, although the STF is a Target Language Compiler (TLC) file, it contains embedded MATLAB code. Before creating or modifying an STF, you should acquire a working knowledge of TLC and of the MATLAB language. “Target Language Compiler” (Simulink Coder) and “Scripts vs. Functions” (MATLAB) describe the features and syntax of both the TLC and MATLAB languages.

While reading this section, you may want to refer to the STFs provided with the code generator. Most of these files are stored in the target-specific folders under *`matlabroot`*`/`

`rtw/c` (open). Additional STFs are stored under *`matlabroot`*`/toolbox/rtw/targets` (open).

System Target File Naming and Location Conventions

An STF must be located in a folder on the MATLAB path for the target to be displayed in the System Target File Browser and invoked in the build process. Follow the location and naming conventions for STFs and related target files given in "Folder and File Naming Conventions" on page 71-11.

System Target File Structure

Overview

This section is a guide to the structure and contents of an STF. The following listing shows the general structure of an STF. Note that this is not a complete code listing of an STF. The listing consists of excerpts from each of the sections that make up an STF.

```
%%----------------------------
%% Header Comments Section
%%----------------------------
%% SYSTLC: Example Real-Time Target
%%    TMF: my_target.tmf MAKE: make_rtw EXTMODE: ext_comm
%% Inital comments contain directives for STF Browser.
%% Documentation, date, copyright, and other info may follow.
          ...
%selectfile NULL_FILE
         ...
%%----------------------------
%% TLC Configuration Variables Section
%%----------------------------
%% Assign code format, language, target type.
%%
%assign CodeFormat = "Embedded-C"
```

```
%assign TargetType = "RT"
%assign Language   = "C"
%%
%%-----------------------------
%% TLC Program Entry Point
%%-----------------------------
%% Call entry point function.
%include "codegenentry.tlc"
%%
%%-----------------------------
%% (OPTIONAL) Generate Files for Build Process
%%-----------------------------
%include "mytarget_genfiles.tlc"
%%-----------------------------
%% RTW_OPTIONS Section
%%-----------------------------
/%
  BEGIN_RTW_OPTIONS
  %% Define rtwoptions structure array. This array defines target-specific
  %% code generation variables, and controls how they are displayed.
  rtwoptions(1).prompt = 'example code generation options';
        ...
  rtwoptions(6).prompt = 'Show eliminated blocks';
  rtwoptions(6).type = 'Checkbox';
         ...
  %---------------------------------------%
  % Configure RTW code generation settings %
  %---------------------------------------%
         ...
  %%---------------------------
  %% rtwgensettings Structure
  %%---------------------------
  %% Define suffix text for naming build folder here.
  rtwgensettings.BuildDirSuffix = '_mytarget_rtw'
  %% Callback compatibility declaration
  rtwgensettings.Version = '1';

  %% (OPTIONAL) target inheritance declaration
  rtwgensettings.DerivedFrom = 'ert.tlc';
  %% (OPTIONAL) other rtwGenSettings fields...
         ...
  END_RTW_OPTIONS
%/
%%-----------------------------
%% targetComponentClass - MATHWORKS INTERNAL USE ONLY
%% REMOVE NEXT SECTION FROM USER_DEFINED CUSTOM TARGETS
%%-----------------------------
/%
  BEGIN_CONFIGSET_TARGET_COMPONENT
  targetComponentClass = 'Simulink.ERTTargetCC';
  END_CONFIGSET_TARGET_COMPONENT
%/
```

If you are creating a custom target based on an existing STF, you must remove the `targetComponentClass` section (bounded by the directives

BEGIN_CONFIGSET_TARGET_COMPONENT and END_CONFIGSET_TARGET_COMPONENT). This section is reserved for the use of targets developed internally by MathWorks.

Header Comments

These lines at the head of the file are formatted as TLC comments. They provide required information to the System Target File Browser and to the build process. Note that you must place the browser comments at the head of the file, before other comments or TLC statements.

The presence of the comments enables the code generator to detect STFs. When the System Target File Browser is opened, the code generator scans the MATLAB path for TLC files that have formatted header comments. The comments contain the following directives:

- SYSTLC: The descriptor that appears in the browser.
- TMF: Name of the template makefile (TMF) to use during build process. When the target is selected, this filename is displayed in the "Template makefile" (Simulink Coder) field of the **Code Generation** pane of the Configuration Parameters dialog box.
- MAKE: `make` command to use during build process. When the target is selected, this command is displayed in the **Make command** field of the **Code Generation** pane of the Configuration Parameters dialog box.
- EXTMODE: Name of external mode interface file (if any) associated with your target. If your target does not support external mode, use `no_ext_comm`.

The following header comments are from *`matlabroot`*`/rtw/c/ert/ert.tlc`.

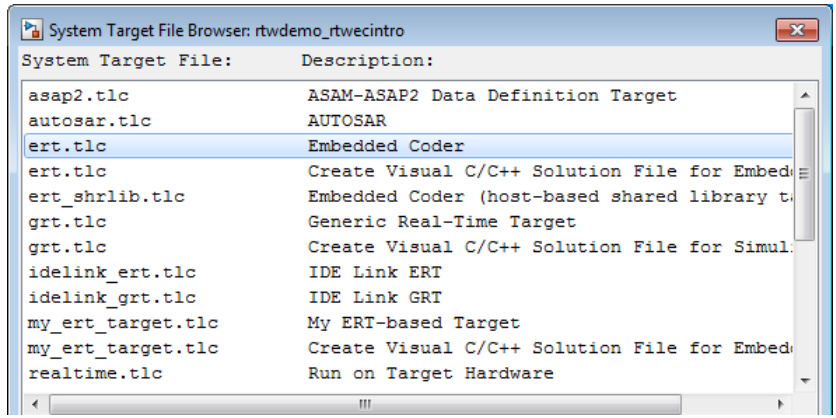
```
%% SYSTLC: Embedded Coder TMF: ert_default_tmf MAKE: make_rtw \
%%    EXTMODE: ext_comm
%% SYSTLC: Create Visual C/C++ Solution File for Embedded Coder\
%%    TMF: RTW.MSVCBuild MAKE: make_rtw EXTMODE: ext_comm
.
.
.
```

Note:

- Limitation: Each comment can only contain a maximum of two lines, as shown in the preceding example.
- If you do not specify the TMF or EXTMODE fields in the system target file, the file is still valid. To change the values for the parameters **Template makefile**

(`TemplateMakefile`) and **External mode** (`ExtMode`), you can instead use the callback function specified by `rtwgensettings.SelectCallback`.

Note that you can specify more than one group of directives in the header comments. Each such group is displayed as a different target configuration in the System Target File Browser. In the above example, the first two lines of code specify the default configuration of the ERT target. The next two lines specify a configuration that creates and builds a Microsoft Visual C++ Solution (`.sln`) file. The figure below shows how these configurations appear in the System Target File Browser.



See “Create a Custom Target Configuration” on page 71-48 for an example of customized header comments.

TLC Configuration Variables

This section of the STF assigns global TLC variables that relate to the overall code generation process.

For an embedded target, in most cases you should simply use the global TLC variable settings used by the ERT target (`ert.tlc`). It is especially important that your STF use the `'Embedded-C'` value for the `CodeFormat` TLC variable and uses the corresponding `rtwgensettings.DerivedFrom = 'ert.tlc'` in the `RTW_OPTIONS` section of the TLC file. Verify that values are assigned to the following variables:

- `CodeFormat`: The `CodeFormat` TLC variable selects generated code features. The `'Embedded-C'` value for this variable is used by the ERT target. Your ERT-based

target should specify `'Embedded-C'` as the value for `CodeFormat`. This selection is designed for production code, minimal memory usage, static memory allocation, and a simplified interface to generated code.

For information on other values for the `CodeFormat` TLC variable, see "Compare System Target File Support" (Simulink Coder).

- `Language`: The only valid value is `C`, which enables support for `C` or `C++` code generation as specified by the configuration parameter `TargetLang`.
- `TargetType`: The code generator defines the preprocessor symbols `RT` and `NRT` to distinguish simulation code from real-time code. These symbols are used in conditional compilation. The `TargetType` variable determines whether `RT` or `NRT` is defined.

  Most targets are intended to generate real-time code. They assign `TargetType` as follows.

  ```
  %assign TargetType = "RT"
  ```

  Some targets, such as the model reference simulation target, accelerated simulation target, RSim target, and S-function target, generate code for use in nonreal time only. Such targets assign `TargetType` as follows.

  ```
  %assign TargetType = "NRT"
  ```

TLC Program Entry Point and Related %includes

The code generation process normally begins with `codegenentry.tlc`. The STF invokes `codegenentry.tlc` as follows.

```
%include "codegenentry.tlc"
```

Note `codegenentry.tlc` and the lower-level TLC files assume that `CodeFormat`, `TargetType`, and `Language` have been assigned. Set these variables before including `codegenentry.tlc`.

If you need to implement target-specific code generation features, you should include the TLC file `mytarget_genfiles.tlc` in your STF. This file provides a mechanism for executing custom TLC code before and after invoking `codegenentry.tlc`. For information on this mechanism, see

- "Using mytarget_genfiles.tlc" on page 71-45 for an example of custom TLC code for execution after the main code generation entry point.
- "Target Development and the Build Process" on page 71-23 for general information on the build process, and for information on other build process customization hooks.

Another way to customize the code generation process is to call lower-level functions (normally invoked by `codegenentry.tlc`) directly, and include your own TLC functions at each stage of the process. This approach should be taken with caution. See "TLC Files" (Simulink Coder) for more information.

The lower-level functions called by `codegenentry.tlc` are

- `genmap.tlc`: maps block names to corresponding language-specific block target files.
- `commonsetup.tlc`: sets up global variables.
- `commonentry.tlc`: starts the process of generating code.

RTW_OPTIONS Section

The `RTW_OPTIONS` section is bounded by the directives:

```
/%
  BEGIN_RTW_OPTIONS
.
.
.
  END_RTW_OPTIONS
%/
```

The first part of the `RTW_OPTIONS` section defines an array of `rtwoptions` structures. This structure is discussed in "Using rtwoptions to Display Custom Target Options" on page 71-38.

The second part of the `RTW_OPTIONS` section defines `rtwgensettings`, a structure defining the build folder name and other settings for the code generation process. See "rtwgensettings Structure" on page 71-35 for information about `rtwgensettings`.

rtwgensettings Structure

The final part of the STF defines the `rtwgensettings` structure. This structure stores information that is written to the *`model`*`.rtw` file and used by the build process. The `rtwgensettings` fields of most interest to target developers are

- `rtwgensettings.Version`: Use this property to enable `rtwoptions` callbacks and to use the Callback API in `rtwgensettings.SelectCallback`.

  Note: To use callbacks, you *must* set:

  ```
  rtwgensettings.Version = '1';
  ```

  Add the statement above to the **Configure RTW code generation settings** section of the system target file.
- `rtwgensettings.DerivedFrom`: This structure field defines the system target file from which options are to be inherited. See "Inheriting Target Options" on page 71-44.
- `rtwgensettings.SelectCallback`: This structure field specifies a `SelectCallback` function. You must set `rtwgensettings.Version = '1';` or your callback will be ignored. `SelectCallback` is associated with the target rather than with any of its individual options. The `SelectCallback` function is triggered when the user selects a target with the System Target File browser.

  The `SelectCallback` function is useful for setting up (or disabling) configuration parameters specific to the target.

  The following code installs a `SelectCallback` function:

  ```
  rtwgensettings.SelectCallback = 'my_select_callback_handler(hDlg,hSrc)';
  ```

  The arguments to the `SelectCallback` function `(hDlg, hSrc)` are handles to private data used by the callback API functions.

  Note If you have developed a custom target and you want it to be compatible with model referencing, you must implement a `SelectCallback` function to declare model reference compatibility. See "Support Model Referencing" on page 71-83.
- `rtwgensettings.ActivateCallback`: this property specifies an `ActivateCallback` function. The `ActivateCallback` function is triggered when the active configuration set of the model changes. This could happen during model loading, and also when the user changes the active configuration set.

  The following code installs an `ActivateCallback` function:

  ```
  rtwgensettings.ActivateCallback = 'my_activate_callback_handler(hDlg,hSrc)';
  ```

The arguments to the `ActivateCallback` function `(hDlg,hSrc)` are handles to private data used by the callback API functions.

- `rtwgensettings.PostApplyCallback`: this property specifies a `PostApplyCallback` function. The `PostApplyCallback` function is triggered when the user clicks the **Apply** or **OK** button after editing options in the Configuration Parameters dialog box. The `PostApplyCallback` function is called after the changes have been applied to the configuration set.

  The following code installs an `PostApplyCallback` function:

  ```
  rtwgensettings.PostApplyCallback = 'my_postapply_callback_handler(hDlg,hSrc)';
  ```

  The arguments to the `PostApplyCallback` function `(hDlg, hSrc)` are handles to private data used by the callback API functions.

- `rtwgensettings.BuildDirSuffix`: Most targets define a folder name suffix that identifies build folders created by the target. The build process appends the suffix defined in the `rtwgensettings.BuildDirSuffix` field to the model name to form the name of the build folder. For example, if you define `rtwgensettings.BuildDirSuffix` as follows

  ```
  rtwgensettings.BuildDirSuffix = '_mytarget_rtw'
  ```

  the build folders are named *`model`*`_mytarget_rtw`.

Additional Code Generation Options

"Configure Generated Code with TLC" (Simulink Coder) describes additional TLC code generation variables. End users of a target can assign these variables by entering a MATLAB command of the form

```
set_param(modelName,'TLCOptions','-aVariable=val');
```

(For more information, see "Specify TLC for Code Generation" (Simulink Coder).)

However, the preferred approach is to assign these variables in the STF using statements of the form:

```
%assign Variable = val
```

For readability, we recommend that you add such assignments in the section of the STF after the comment **Configure RTW code generation settings**.

Model Reference Considerations

See “Support Model Referencing” on page 71-83 for important information on STF and other modifications you may need to make to support the code generator model referencing features.

Define and Display Custom Target Options

Using rtwoptions to Display Custom Target Options

You control the options to display in the **Code Generation** pane of the Configuration Parameters dialog box by customizing the `rtwoptions` structure in your system target file.

The fields of the `rtwoptions` structure define variables and associated user interface elements to be displayed in the Configuration Parameters dialog box. Using the `rtwoptions` structure array, you can define target-specific options displayed in the dialog box and organize options into categories. You can also write callback functions to specify how these options are processed.

When the **Code Generation** pane opens, the `rtwoptions` structure array is scanned and the listed options are displayed. Each option is represented by an assigned user interface element (check box, edit field, menu, or push button), which displays the current option value.

The user interface elements can be in an enabled or disabled (grayed-out) state. If an option is enabled, the user can change the option value.

You can also use the `rtwoptions` structure array to define special NonUI elements that cause callback functions to be executed, but that are not displayed in the **Code Generation** pane. See “NonUI Elements” on page 71-42 for details.

The elements of the `rtwoptions` structure array are organized into groups. Each group of items begins with a header element of type `Category`. The default field of a `Category` header must contain a count of the remaining elements in the category.

The `Category` header is followed by options to be displayed on the **Code Generation** pane. The header in each category is followed by one or more option definition elements.

Each category of target options corresponds to options listed under **Code Generation** in the Configuration Parameters dialog box.

The table rtwoptions Structure Fields Summary summarizes the fields of the `rtwoptions` structure.

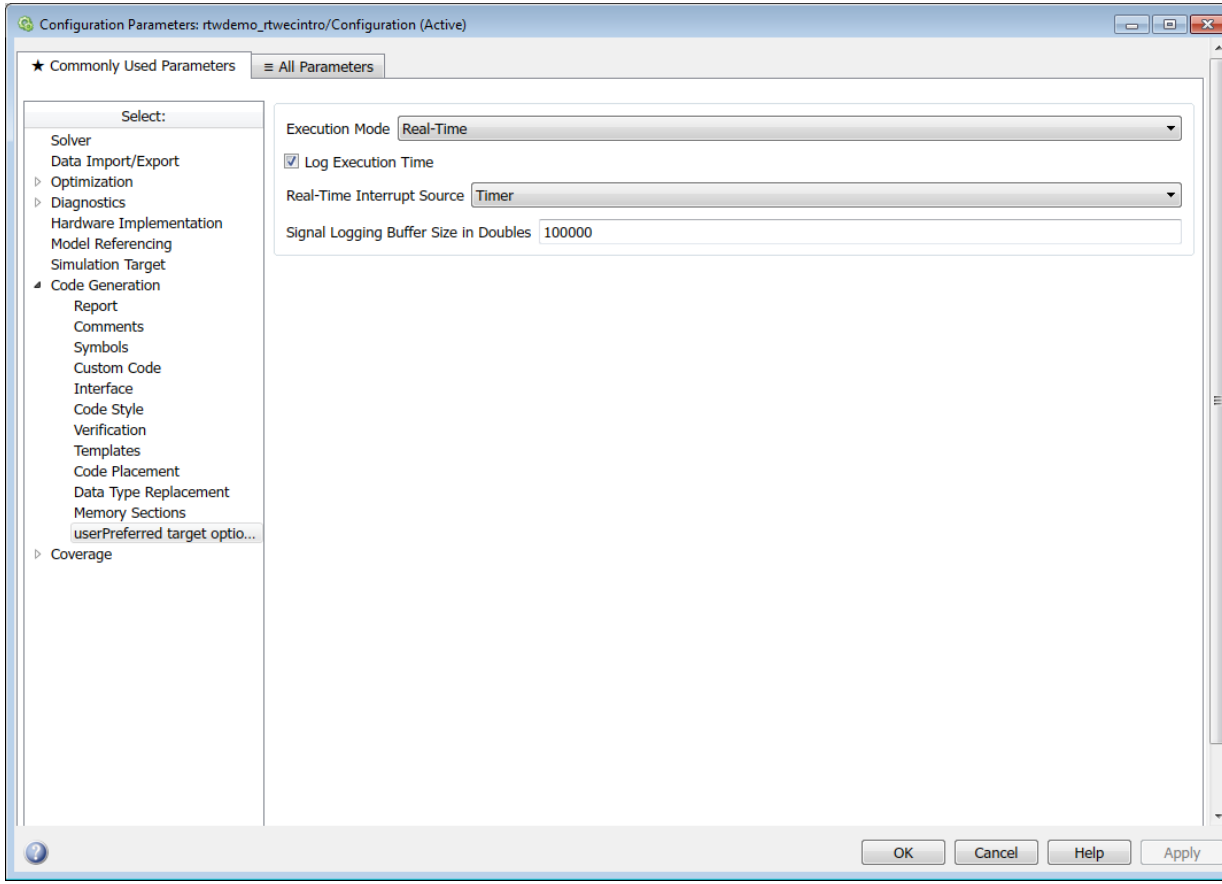
Example rtwoptions Structure

The following `rtwoptions` structure is excerpted from an example system target file, *`matlabroot`*`/toolbox/rtw/rtwdemos/rtwoptions_demo/usertarget.tlc`. The code defines an `rtwoptions` structure array. The default field of the first (header) element is set to `4`, indicating the number of elements that follow the header.

```
  rtwoptions(1).prompt        = 'userPreferred target options (I)';
  rtwoptions(1).type          = 'Category';
  rtwoptions(1).enable        = 'on';
  rtwoptions(1).default       = 4;   % number of items under this category
                                     % excluding this one.
  rtwoptions(1).popupstrings  = '';  % At the first item, user has to
  rtwoptions(1).tlcvariable   = '';  % initialize all supported fields
  rtwoptions(1).tooltip       = '';
  rtwoptions(1).callback      = '';
  rtwoptions(1).makevariable  = '';

  rtwoptions(2).prompt        = 'Execution Mode';
  rtwoptions(2).type          = 'Popup';
  rtwoptions(2).default       = 'Real-Time';
  rtwoptions(2).popupstrings  = 'Real-Time|UserDefined';
  rtwoptions(2).tlcvariable   = 'tlcvariable1';
  rtwoptions(2).tooltip       = ['See this text as tooltip'];

  rtwoptions(3).prompt        = 'Log Execution Time';
  rtwoptions(3).type          = 'Checkbox';
  rtwoptions(3).default       = 'on';
  rtwoptions(3).tlcvariable   = 'RL32LogTETModifier';
  rtwoptions(3).tooltip       = ['']; % no tooltip

  rtwoptions(4).prompt        = 'Real-Time Interrupt Source';
  rtwoptions(4).type          = 'Popup';
  rtwoptions(4).default       = 'Timer';
  rtwoptions(4).popupstrings  = 'Timer|5|6|7|8|9|10|11|12|13|14|15';
  rtwoptions(4).tlcvariable   = 'tlcvariable3';
  rtwoptions(4).callback      = 'usertargetcallback(hDlg, hSrc, ''tlcvariable3'')';
  rtwoptions(4).tooltip       = [''];
  rtwoptions(4).tooltip       = ['See TLC file for how to use reserved '...
    ' keyword ''hDlg'', and ''hSrc''.'];
...
  rtwoptions(5).prompt        = 'Signal Logging Buffer Size in Doubles';
  rtwoptions(5).type          = 'Edit';
```

```
rtwoptions(5).default       = '100000';
rtwoptions(5).tlcvariable   = 'tlcvariable2';
rtwoptions(5).tooltip       = [''];
```

The first element adds a **userPreferred target options (I)** pane under **Code Generation** in the Configuration Parameters dialog box. The pane displays the options defined in `rtwoptions(2)`, `rtwoptions(3)`, `rtwoptions(4)`, and `rtwoptions(5)`.



If you want to define a large number of options, you can define multiple `Category` groups within a single system target file.

Note the `rtwoptions` structure and callbacks are written in MATLAB code, although they are embedded in a TLC file. To verify the syntax of your `rtwoptions` structure

definitions and code, you can execute the commands at the MATLAB prompt by copying and pasting them to the MATLAB Command Window.

To learn more about `usertarget.tlc` and the example callback files provided with it, see "Example System Target File With Customized rtwoptions" on page 71-43. For more examples of target-specific `rtwoptions` definitions, see the *`target`*`.tlc` files under *`matlabroot`*`/rtw/c` (open).

rtwoptions Structure Fields Summary lists the fields of the `rtwoptions` structure.

rtwoptions Structure Fields Summary

Field Name	Description
`callback`	For examples of callback usage, see "Example System Target File With Customized rtwoptions" on page 71-43.
`closecallback` `(obsolete)`	Do not use `closecallback`. Use `rtwgensettings.PostApplyCallback` instead (see "rtwgensettings Structure" on page 71-35). `closecallback` is ignored. For examples of callback usage, see "Example System Target File With Customized rtwoptions" on page 71-43.
`default`	Default value of the option (empty if the `type` is `Pushbutton`).
`enable`	Must be `'on'` or `'off'`. If `'on'`, the option is displayed as an enabled item; otherwise, as a disabled item.
`makevariable`	Template makefile token (if any) associated with the option. The `makevariable` is expanded during processing of the template makefile. See "Template Makefile Tokens" on page 71-62.
`modelReferenceParameter-Check`	Specifies whether the option must have the same value in a referenced model and its parent model. If this field is unspecified or has the value `'on'` the option values must be same. If the field is specified and has the value `'off'` the option values can differ. See "Controlling Configuration Option Value Agreement" on page 71-88.
`NonUI`	Element that is not displayed, but is used to invoke a close or open callback. See "NonUI Elements" on page 71-42.
`opencallback`	Do not use `opencallback`.

Field Name	Description
`(obsolete)`	Use `rtwgensettings.SelectCallback` instead (see "rtwgensettings Structure" on page 71-35). For examples of callback usage, see "Example System Target File With Customized rtwoptions" on page 71-43.
`popupstrings`	If `type` is `Popup`, `popupstrings` defines the items in the menu. Items are delimited by the `"|"` (vertical bar) character. The following example defines the items of the **MAT-file variable name modifier** menu used by the GRT target. `'rt_|_rt|none'`
`prompt`	Label for the option.
`tlcvariable`	Name of TLC variable associated with the option.
`tooltip`	Help text displayed when mouse is over the item.
`type`	Type of element: `Checkbox`, `Edit`, `NonUI`, `Popup`, `Pushbutton`, or `Category`.

NonUI Elements

Elements of the `rtwoptions` array that have type `NonUI` exist solely to invoke callbacks. A `NonUI` element is not displayed in the Configuration Parameters dialog box. You can use a `NonUI` element if you want to execute a callback that is not associated with a user interface element, when the dialog box opens or closes. See the next section, "Example System Target File With Customized rtwoptions" on page 71-43 for an example.

Note: The default value of a `NonUI` element determines the set of values allowed for that element.

- If the default value is `'0'` or `'1'`, the element stores a Boolean value.
- If the default value contains an integer other than `'0'` or `'1'`, the element stores a value of type `int32`.
- If the default value does not contain an integer, the element is evaluated as a character vector.

Example System Target File With Customized rtwoptions

A working system target file, with MATLAB file callback functions, has been provided as an example of how to use the `rtwoptions` structure to display and process custom options on the **Code Generation** pane. The examples are compatible with the callback API.

The example target files are in the folder (open):

`matlabroot``/toolbox/rtw/rtwdemos/rtwoptions_demo`

The example target files include:

- `usertarget.tlc`: The example system target file. This file illustrates how to define custom menus, check boxes, and edit fields. The file also illustrates the use of callbacks.
- `usertargetcallback.m`: A MATLAB file callback invoked by a menu.

Refer to the example files while reading this section. The example system target file, `usertarget.tlc`: illustrates the use of `rtwoptions` to display the following custom target options:

- The **Execution Mode** menu.
- The **Log Execution Time** check box.
- The **Real-Time Interrupt Source** menu. The menu executes a callback defined in an external file, `usertargetcallback.m`. The TLC variable associated with the menu is passed in to the callback, which displays the menu's current value.
- The edit field **Signal Logging Buffer Size in Doubles**.

Try studying the example code while interacting with the example target options in the Configuration Parameters dialog box. To interact with the example target file,

1. Make *`matlabroot`*`/toolbox/rtw/rtwdemos/rtwoptions_demo` (open) your working folder.
2. Open a model of your choice.
3. Open the Configuration Parameters dialog box and select the **Code Generation** pane.
4. Click **Browse**. The System Target File Browser opens. Select `usertarget.tlc`. Then click **OK**.
5. Observe that the **Code Generation** pane contains a custom sub-tab: **userPreferred target options (I)**.

6 As you interact with the options in this category and open and close the Configuration Parameters dialog box, observe the messages displayed in the MATLAB Command Window. These messages are printed from code in the STF, or from callbacks invoked from the STF.

Inheriting Target Options

`ert.tlc` provides a basic set of Embedded Coder code generation options. If your target is based on `ert.tlc`, your STF should normally inherit the options defined in ERT.

Use the `rtwgensettings.DerivedFrom` field in the `rtwgensettings` structure to define the system target file from which options are to be inherited. You should convert your custom target to use this mechanism as follows.

Set the `rtwgensettings.DerivedFrom` field value as in the following example:

```
rtwgensettings.DerivedFrom = 'stf.tlc';
```

where `stf` is the name of the system target file from which options are to be inherited. For example:

```
rtwgensettings.DerivedFrom = 'ert.tlc';
```

When the Configuration Parameters dialog box executes this line of code, it includes the options from `stf.tlc` automatically. If `stf.tlc` is a MathWorks internal system target file that has been converted to a new layout, the dialog box displays the inherited options using the new layout.

Handling Unsupported Options

If your target does not support all of the options inherited from `ert.tlc`, you should detect unsupported option settings and display a warning or error message. In some cases, if a user has selected an option your target does not support, you may need to abort the build process. For example, if your target does not support the **Generate an example main program** option, the build process should not be allowed to proceed if that option is selected.

Even though your target may not support all inherited ERT options, it is required that the ERT options are retained in the **Code Generation** pane of the Configuration Parameters dialog box. Do not simply remove unsupported options from the `rtwoptions` structure in the STF. Options must be in the dialog box to be scanned by the code generator when it performs optimizations.

For example, you may want to prevent users from turning off the **Single output/update function** option. It may seem reasonable to remove this option from the dialog box and simply assign the TLC variable `CombineOutputUpdateFcns` to `on`. However, if the option is not included in the dialog box, the code generator assumes that output and update functions are *not* to be combined. Less efficient code is generated as a result.

Tips and Techniques for Customizing Your STF

Introduction

The following sections include information on techniques for customizing your STF, including

- How to invoke custom TLC code from your STF
- Approaches to supporting multiple development environments

Required and Recommended %includes

If you need to implement target-specific code generation features, we recommend that your STF include the TLC file `mytarget_genfiles.tlc`.

Once your STF has set up the required TLC environment, you must include `codegenentry.tlc` to start the standard code generation process.

`mytarget_genfiles.tlc` provides a mechanism for executing custom TLC code after the main code generation entry point. See "Using mytarget_genfiles.tlc" on page 71-45.

Using mytarget_genfiles.tlc

`mytarget_genfiles.tlc` (optional) is useful as a central file from which to invoke target-specific TLC files that generate additional files as part of your target build process. For example, your target may create sub-makefiles or project files for a development environment, or command scripts for a debugger to do automatic downloads.

The build process can then invoke these generated files either directly from the make process, or after the executable is created. This is done with the *STF*_make_rtw_hook.m mechanism, as described in "Customize Build Process with STF_make_rtw_hook File" (Simulink Coder).

The following TLC code shows an example mytarget_genfiles.tlc file.

```
%selectfile NULL_FILE

%assign ModelName = CompiledModel.Name

%% Create Debugger script
%assign model_script_file = "%<ModelName>.cfg"
%assign script_file = "debugger_script_template.tlc"

%if RTWVerbose
   %selectfile STDOUT
   ### Creating %<model_script_file>
   %selectfile NULL_FILE
%endif

%include "%<script_file>"
%openfile bld_file = "%<model_script_file>"
%<CreateDebuggerScript()>
%closefile bld_file
```

Handling Aliases for Target Option Values

This section describes utility functions that can be used to detect and resolve alias values or legacy values when testing user-specified values for the target device type (ProdHWDeviceType) and the code replacement library (CodeReplacementLibrary).

RTW.isHWDeviceTypeEq

To test if two target device type values represent the same hardware device, invoke the following function:

```
result = RTW.isHWDeviceTypeEq(type1,type2)
```

where *type1* and *type2* are character vectors containing target device type values or aliases.

The RTW.isHWDeviceTypeEq function returns true if *type1* and *type2* are character vectors representing the same hardware device. For example, the following call returns true:

```
RTW.isHWDeviceTypeEq('Specified','Generic->Custom')
```

For a description of the target device type option `ProdHWDeviceType`, see the command-line information for the **Hardware Implementation** pane parameters "Device vendor" (Simulink) and "Device type" (Simulink).

RTW.resolveHWDeviceType

To return the device type value for a hardware device, given a value that might be an alias or legacy value, invoke the following function:

```
result = RTW.resolveHWDeviceType(type)
```

where *`type`* is a character vector containing a target device type value or alias.

The `RTW.resolveHWDeviceType` function returns the device type value of the device. For example, the following calls both return `'Generic->Custom'`:

```
RTW.resolveHWDeviceType('Specified')
RTW.resolveHWDeviceType('Generic->Custom')
```

For a description of the target device type option `ProdHWDeviceType`, see the command-line information for the **Hardware Implementation** pane parameters "Device vendor" (Simulink) and "Device type" (Simulink).

RTW.isTflEq

To test if two code replacement library (CRL) names represent the same CRL, invoke the following function:

```
result = RTW.isTflEq(name1,name2)
```

where *`name1`* and *`name2`* are character vectors containing CRL values or aliases.

The `RTW.isTflEq` function returns true if *`name1`* and *`name2`* are character vectors representing the same code replacement library. For example, the following call returns true:

```
RTW.isTflEq('GNU','GNU C99 extensions')
```

For a description of the `CodeReplacementLibrary` parameter, see "Code replacement library" (Simulink Coder).

RTW.resolveTflName

To return the CRL value for a code replacement library, given a value that might be an alias or legacy value, invoke the following function:

```
result = RTW.resolveTflName(name)
```

where *name* is a character vector containing a CRL value or alias.

The `RTW.resolveTflName` function returns the value of the referenced code replacement library. For example, the following calls both return `'GNU C99 extensions'`:

```
RTW.resolveTflName('GNU')
RTW.resolveTflName('GNU C99 extensions')
```

For a description of the `CodeReplacementLibrary` parameter, see "Code replacement library" (Simulink Coder).

Supporting Multiple Development Environments

Your target may require support for multiple development environments (for example, two or more cross-compilers) or multiple modes of code generation (for example, generating a binary executable versus generating a project file for your compiler).

One approach to this requirement is to implement multiple STFs. Each STF invokes a template makefile for the development environment. This amounts to providing two separate targets.

Create a Custom Target Configuration

Introduction

This tutorial can supplement the example target guides described in "Sample Custom Targets" on page 71-9. For an introduction and example files, try the example targets first.

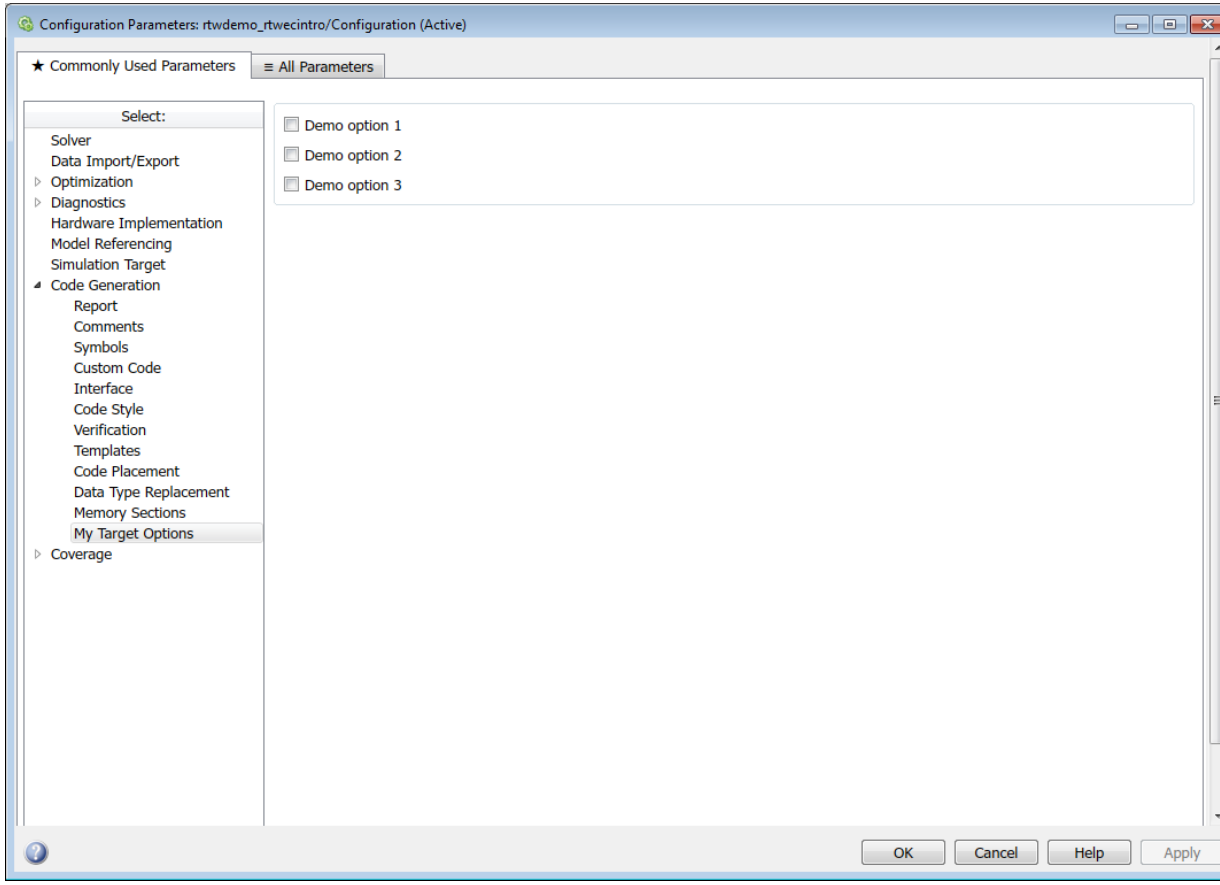
This tutorial guided you through the process of creating an ERT-based target, `my_ert_target`. This exercise illustrates several tasks, which are typical for creating a custom target:

- Setting up target folders and modifying the MATLAB path.
- Making modifications to a standard STF and TMF such that the custom target is visible in the System Target File Browser, inherits ERT options, displays target-specific options, and generates code with the default host-based compiler.
- Testing the build process with the custom target, using a simple model that incorporates an inlined S-function.

During this exercise, you implement an operational, but skeletal, ERT-based target. This target can be useful as a starting point in a complete implementation of a custom embedded target.

my_ert_target Overview

In the following sections, you create a skeletal target, `my_ert_target`. The target inherits and supports the standard options of the ERT target and displays additional target-specific options in the Configuration Parameters dialog box (see Target-Specific Options for my_ert_target).



Target-Specific Options for my_ert_target

`my_ert_target` supports a toolchain-based build, generating code and executables that run on the host system. `my_ert_target` uses the `lcc` compiler on a Microsoft Windows platform. The chosen compiler is readily available and is distributed with the code generator. On a Microsoft Windows platform, if you use a different compiler, you can set up `lcc` temporarily as your default compiler through the following MATLAB command:

```
mex -setup
```

The software displays links for supported compilers that are installed on your computer. Click the `lcc` link.

Note On Linux systems, make sure that you have an installed C compiler. If so, you can use Linux folder syntax to complete this exercise.

`my_ert_target` can also support template makefile-based builds. For more information about using this target with the template makefile approach, see "Create ERT-Based TMF" on page 71-58.

You can test `my_ert_target` with a model that is compatible with the ERT target (see "Select a System Target File" on page 30-2). Generated programs operate identically to ERT generated programs.

To simplify the testing of your target, test with `targetmodel`, a very simple fixed-step model (see "Create Test Model and S-Function" on page 71-58). The S-Function block in `targetmodel` uses the source code from the `timestwo` example, and generates fully inlined code. See "S-Function Examples" (Simulink) and "Inline S-Functions with TLC" (Simulink Coder) for further discussion of the `timestwo` example S-function.

Creating Target Folders

Create folders to store the target files and add them to the MATLAB path, following the recommended conventions (see "Folder and File Naming Conventions" on page 71-11). You also create a folder to store the test model, S-function, and generated code.

This example assumes that your target and model folders are located within the folder `c:/work`. Do not place your target and model folders within the MATLAB folder tree (that is, in or under the *`matlabroot`* folder).

To create the folders and make them accessible:

1. Create a target root folder, `my_ert_target`. From the MATLAB Command Window on a Windows platform, enter:

   ```
   cd c:/work
   mkdir my_ert_target
   ```

2. Within the target root folder, create a subfolder to store your target files.

   ```
   mkdir my_ert_target/my_ert_target
   ```

3. Add these folders to your MATLAB path.

   ```
   addpath c:/work/my_ert_target
   addpath c:/work/my_ert_target/my_ert_target
   ```

4. Create a folder, `my_targetmodel`, to store the test model, S-function, and generated code.

```
mkdir my_targetmodel
```

Create ERT-Based, Toolchain Compliant STF

Create an STF for your target by copying and modifying the standard STF for the ERT target. Then, validate the STF by viewing the new target in the System Target File Browser and in the Configuration Parameters dialog box.

Editing the STF

To edit the STF, use these steps:

1. Change your working folder to the folder you created in "Creating Target Folders" on page 71-51.

   ```
   cd c:/work/my_ert_target/my_ert_target
   ```

2. Place a copy of *`matlabroot`*`/rtw/c/ert/ert.tlc` in `c:/work/my_ert_target/my_ert_target` and rename it to `my_ert_target.tlc`. The file `ert.tlc` is the STF for the ERT target.

3. Open `my_ert_target.tlc` in a text editor of your choice.

4. Customize the STF, replacing the header comment lines with directives that make your STF visible in the System Target File Browser and define the associated TMF, `make` command, and external mode interface file (if any). For more information about these directives, see "Header Comments" on page 71-32 .

   Replace the header comments in `my_ert_target.tlc` with the following header comments.

   ```
   %% SYSTLC: My ERT-based Target TMF: my_ert_target_lcc.tmf MAKE: make_rtw \
   %%    EXTMODE: no_ext_comm
   ```

5. The file `my_ert_target.tlc` inherits the standard ERT options, using the mechanism described in "Inheriting Target Options" on page 71-44. Therefore, the existing `rtwoptions` structure definition is superfluous. Edit the RTW_OPTIONS section such that it includes only the following code.

   ```
   /%
     BEGIN_RTW_OPTIONS

     %----------------------------------------%
     % Configure RTW code generation settings %
     %----------------------------------------%

     rtwgensettings.BuildDirSuffix = '_ert_rtw';
   ```

```
  END_RTW_OPTIONS
%/
```

6 Delete the code after the end of the `RTW_OPTIONS` section, which is delimited by the directives `BEGIN_CONFIGSET_TARGET_COMPONENT and END_CONFIGSET_TARGET_COMPONENT`. This code is for use only by internal MathWorks developers.

7 Modify the build folder suffix in the `rtwgenSettings` structure in accordance with the conventions described in "rtwgensettings Structure" on page 71-35.

To set the suffix to a character vector for the `_my_ert_target` custom target, change the line

```
rtwgensettings.BuildDirSuffix = '_ert_rtw'
```

to

```
rtwgensettings.BuildDirSuffix = '_my_ert_target_rtw'
```

8 Modify the `rtwgenSettings` structure to inherit options from the ERT target and declare Release 14 or later compatibility as described in "rtwgensettings Structure" on page 71-35. Add the following code to the `rtwgenSettings` definition:

```
rtwgensettings.DerivedFrom = 'ert.tlc';
rtwgensettings.Version = '1';
```

9 Add an `rtwoptions` structure that defines a target-specific options category with three check boxes just after the `BEGIN_RTW_OPTIONS` directive. The following code shows the complete `RTW_OPTIONS` section, including the previous `rtwgenSettings` changes.

```
/%
  BEGIN_RTW_OPTIONS

  rtwoptions(1).prompt         = 'My Target Options';
  rtwoptions(1).type           = 'Category';
  rtwoptions(1).enable         = 'on';
  rtwoptions(1).default        = 3;   % number of items under this category
                                      % excluding this one.
  rtwoptions(1).popupstrings  = '';
  rtwoptions(1).tlcvariable   = '';
  rtwoptions(1).tooltip       = '';
  rtwoptions(1).callback      = '';
  rtwoptions(1).makevariable  = '';

  rtwoptions(2).prompt         = 'Demo option 1';
  rtwoptions(2).type           = 'Checkbox';
  rtwoptions(2).default        = 'off';
  rtwoptions(2).tlcvariable    = 'DummyOpt1';
  rtwoptions(2).makevariable   = '';
```

```
    rtwoptions(2).tooltip        = ['Demo option1 (non-functional)'];
    rtwoptions(2).callback       = '';

    rtwoptions(3).prompt         = 'Demo option 2';
    rtwoptions(3).type           = 'Checkbox';
    rtwoptions(3).default        = 'off';
    rtwoptions(3).tlcvariable    = 'DummyOpt2';
    rtwoptions(3).makevariable   = '';
    rtwoptions(3).tooltip        = ['Demo option2 (non-functional)'];
    rtwoptions(3).callback       = '';

    rtwoptions(4).prompt         = 'Demo option 3';
    rtwoptions(4).type           = 'Checkbox';
    rtwoptions(4).default        = 'off';
    rtwoptions(4).tlcvariable    = 'DummyOpt3';
    rtwoptions(4).makevariable   = '';
    rtwoptions(4).tooltip        = ['Demo option3 (non-functional)'];
    rtwoptions(4).callback       = '';

    %---------------------------------------%
    % Configure RTW code generation settings %
    %---------------------------------------%

    rtwgensettings.BuildDirSuffix = '_my_ert_target_rtw';
    rtwgensettings.DerivedFrom = 'ert.tlc';
    rtwgensettings.Version = '1';
    rtwgensettings.SelectCallback = 'enableToolchainCompliant(hSrc, hDlg)';
    %SelectCallback provides toolchain approach support, but requires custom function
    %Omit this SelectCallback if using the template makefile approach

    END_RTW_OPTIONS
  %/
```

10 Save your changes to `my_ert_target.tlc` and close the file.

Create ToolchainCompliant Function

To enable builds using the toolchain approach, you create a function that corresponds to the `SelectCallback` near the end of the custom STF. This function sets properties for toolchain compliance.

```
function enableToolchainCompliant(hSrc, hDlg)
  hCS = hSrc.getConfigSet();

  % The following parameters enable toolchain compliance.
  slConfigUISetVal(hDlg, hSrc, 'UseToolchainInfoCompliant', 'on');
  hCS.setProp('GenerateMakefile','on');

  % The following parameters are not required for toolchain compliance.
  % But, it is recommended practice to set these default values and
  % disable the parameters (as shown).
  hCS.setProp('RTWCompilerOptimization','off');
  hCS.setProp('MakeCommand','make_rtw');
  hCS.setPropEnabled('RTWCompilerOptimization',false);
```

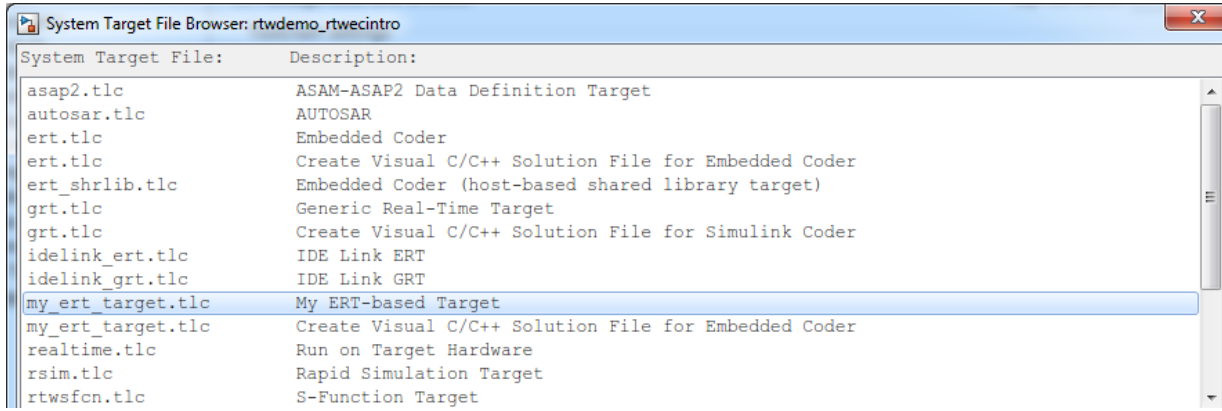
```
  hCS.setPropEnabled('MakeCommand',false);
end
```

Note: If you are using the template makefile approach, omit calling the function enabling toolchain-compliance from your STF file. Instead, use the information in "Create ERT-Based TMF" on page 71-58.

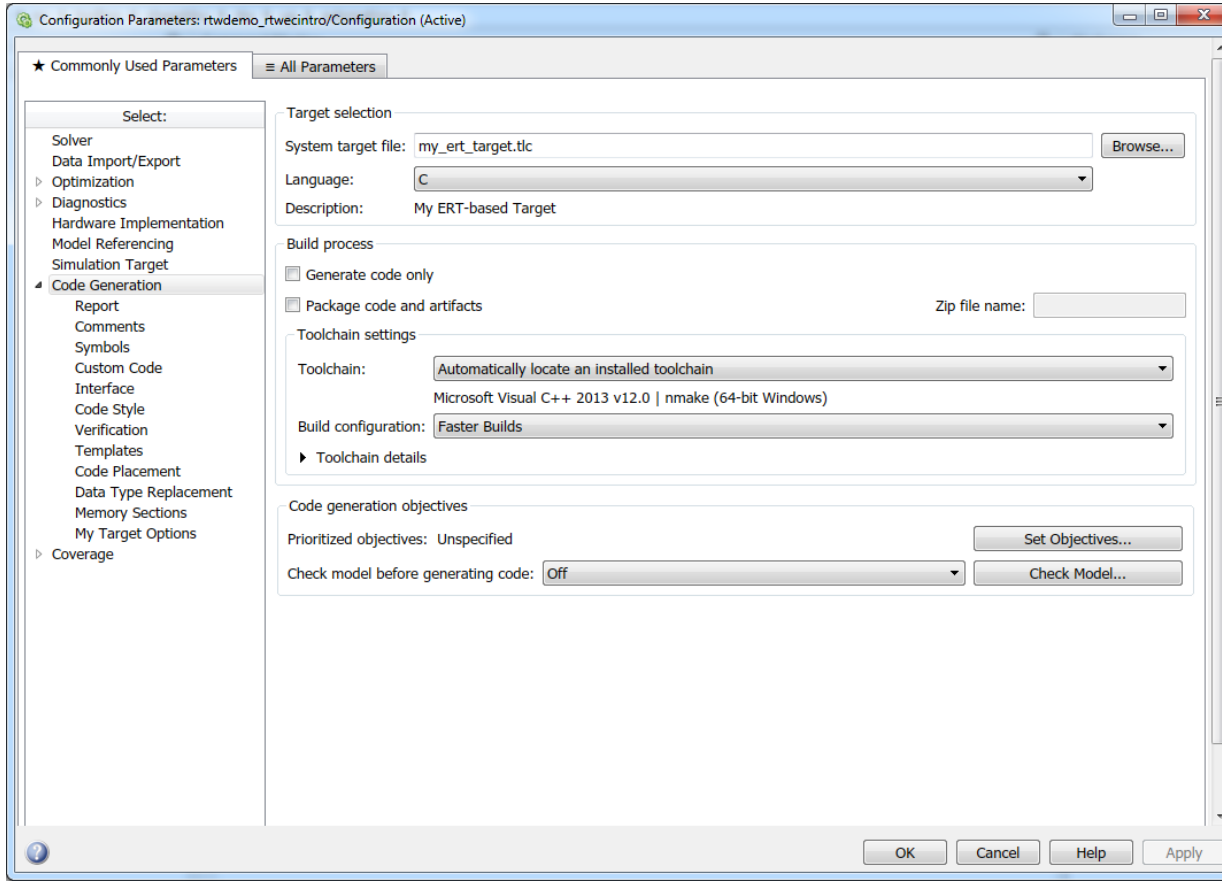
Viewing the STF

At this point, you can verify that the target inherits and displays ERT options as follows:

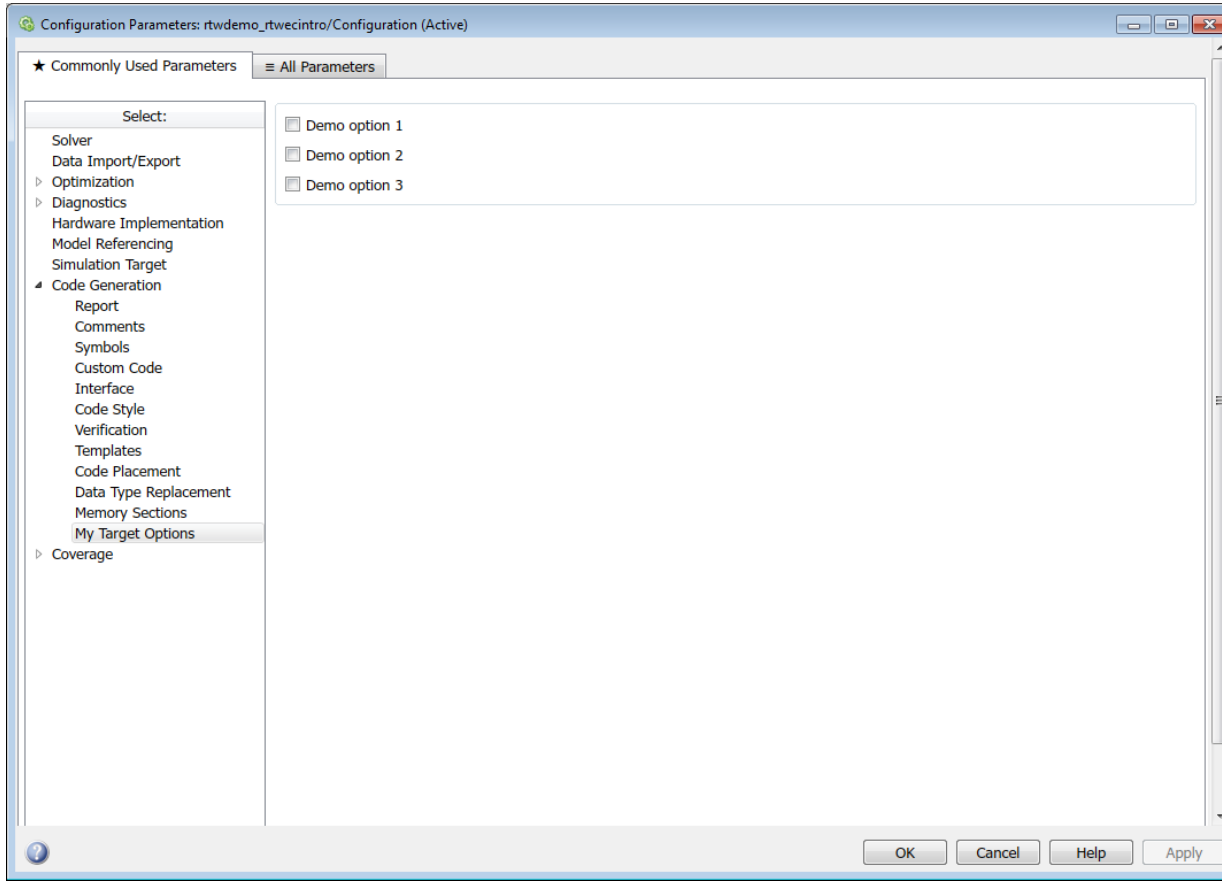
1. Create a new model.
2. Open the Model Explorer or the Configuration Parameters dialog box.
3. Select the **Code Generation** pane.
4. Click **Browse** to open the System Target File browser.
5. In the file browser, scroll through the list of targets to find the new target, `my_ert_target.tlc`. (This step assumes that your MATLAB path contains `c:/work/my_ert_target/my_ert_target`, as previously set in "Creating Target Folders" on page 71-51.)
6. Select `My ERT-based Target` and click **OK.**



7. The **Code Generation** pane now shows that the model is configured for the `my_ert_target.tlc` target. The **System target file**, **Language**, **Toolchain**, and **Build configuration** fields should appear:



8 Select the **My Target Options** pane. The target displays the three check box options defined in the `rtwoptions` structure.



9 Select the **Code Generation** pane and reopen the System Target File Browser.

10 Select the Embedded Coder target (`ert.tlc`). The target displays the standard ERT options.

11 Close the model. You do not need to save it.

The STF for the skeletal target is complete. If you are using the toolchain approach, you are ready to invoke the build process for your target.

If you prefer to use the template makefile approach, the reference to a TMF, `my_ert_target_lcc.tmf`, in the STF header comments prevents you from invoking

the build process for your target until the TMF file is in place. First, you must create a `my_ert_target_lcc.tmf` file.

Create ERT-Based TMF

If you are using the toolchain makefile approach with a toolchain compliant custom target, omit the steps that apply to the template makefile approach. (Skip this section.)

If you are using the templated makefile approach, follow the steps applying to TMF and omit calling the function enabling toolchain-compliance from your STF file, which is described in "Create ERT-Based, Toolchain Compliant STF" on page 71-52.

Create a TMF for your target by copying and modifying the standard ERT TMF for the LCC compiler:

1. Make sure that your working folder is still set to the target file folder you created previously in "Creating Target Folders" on page 71-51.

   ```
   c:/work/my_ert_target/my_ert_target
   ```

2. Place a copy of *`matlabroot`*`/rtw/c/ert/ert_lcc.tmf` in `c:/work/my_ert_target/my_ert_target` and rename it `my_ert_target_lcc.tmf`. The file `ert_lcc.tmf` is the ERT compiler-specific template makefile for the LCC compiler.
3. Open `my_ert_target_lcc.tmf` in a text editor.
4. Change the `SYS_TARGET_FILE` parameter so that the file reference for your `.tlc` file is generated in the `make` file. Change the line

   ```
   SYS_TARGET_FILE = any
   ```

   to

   ```
   SYS_TARGET_FILE = my_ert_target.tlc
   ```

5. Save changes to `my_ert_target_lcc.tmf` and close the file.

Your target can now generate code and build a host-based executable. In the next sections, you create a test model and test the build process using `my_ert_target`.

Create Test Model and S-Function

In this section, you build a simple test model for later use in code generation:

1. Set your working folder to `c:/work/my_targetmodel`.

```
cd c:/work/my_targetmodel
```

For the remainder of this tutorial, `my_targetmodel` is assumed to be the working folder. Your target writes the output files of the code generation process into a build folder within the working folder. When inlined code is generated for the `timestwo` S-function, the build process looks for the TLC implementation of the S-function in the working folder.

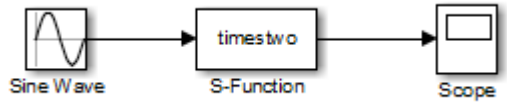
2. Copy the following C and TLC files for the `timestwo` S-function to your working folder:

   - *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/timestwo.c`
   - *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/tlc_c/timestwo.tlc`

3. Build the `timestwo` MEX-file in `c:/work/my_targetmodel`.

   ```
   mex timestwo.c
   ```

4. Create the following model, using an S-Function block from the Simulink User-Defined Functions library. Save the model in your working folder as `targetmodel`.



5. Double-click the S-Function block to open the Block Parameters dialog box. Enter the S-function name `timestwo`. Click **OK**. The block is now bound to the `timestwo` MEX-file.
6. Open Model Explorer or the Configuration Parameters dialog box and select the **Solver** pane.
7. Set the solver **Type** to `fixed-step` and click **Apply**.
8. Save the model.
9. Open the scope and run a simulation. Verify that the `timestwo` S-function multiplies its input by 2.0.

Keep the `targetmodel` model open for use in the next section, in which you generate code using the test model.

Verify Target Operation

In this section you configure `targetmodel` for the `my_ert_target` custom target, and use the target to generate code and build an executable:

1. Open the Configuration Parameters dialog box and select the **Code Generation** pane.
2. Click **Browse** to open the System Target File Browser.
3. In the Browser, select `My ERT-based Target` and click **OK**.
4. The Configuration Parameters dialog box now displays the **Code Generation** pane for `my_ert_target`.
5. Select the **Code Generation** > **Report** pane and select the **Create code generation report** option.
6. Click **Apply** and save the model. The model is configured for `my_ert_target`.
7. Build the model. If the build succeeds, the MATLAB Command Window displays the message below.

   ```
   ### Created executable: ../targetmodel.exe
   ### Successful completion of build procedure for model:
   targetmodel
   ```

   Your working folder contains the `targetmodel.exe` file and the build folder, `targetmodel_my_ert_target_rtw`, which contains generated code and other files. The working folder also contains an `slprj` folder, used internally by the build process.

   The code generator also creates and displays a code generation report.
8. To view the generated model code, go to the code generation report window. In the **Contents** pane, click the `targetmodel.c` link.
9. In `targetmodel.c`, locate the model step function, `targetmodel_step`. Observe the following code.

   ```
   /* S-Function Block: <Root>/S-Function */
   /* Multiply input by two */
   targetmodel_B.SFunction = targetmodel_B.SineWave * 2.0;
   ```

   The presence of this code confirms that the `my_ert_target` custom target has generated an inlined output computation for the S-Function block in the model.

More About

- “About Embedded Target Development” (Simulink Coder)

- “Support Toolchain Approach with Custom Target” (Simulink Coder)
- “Support Model Referencing” (Simulink Coder)
- “Support Compiler Optimization Level Control” (Simulink Coder)
- “Support C Function Prototype Control” (Simulink Coder)
- “Support C++ Class Interface Control” (Simulink Coder)
- “Support Concurrent Execution of Multiple Tasks” (Simulink Coder)

Customize Template Makefiles

To configure or customize a template makefile (TMF), you should be familiar with how the `make` command works and how it processes makefiles. You should also understand makefile build rules. For information on these topics, refer to the documentation provided with the make utility you use.

In this section...
"Template Makefiles and Tokens" on page 71-62
"Invoke the make Utility" on page 71-68
"Structure of the Template Makefile" on page 71-69
"Customize and Create Template Makefiles" on page 71-73

Template Makefiles and Tokens

TMFs are made up of statements containing tokens. The build process expands tokens and creates a makefile, *`model`*`.mk`. TMFs are designed to generate makefiles for specific compilers on specific platforms. The generated *`model`*`.mk` file is tailored to compile and link code generated from your model, using commands specific to your development system.

Creation of model.mk

Template Makefile Tokens

The `make_rtw` command (or a different command provided with some targets) directs the process of generating *`model`*`.mk`. The `make_rtw` command processes the TMF specified on the **Code Generation** pane of the Configuration Parameters dialog box. `make_rtw` copies the TMF, line by line, expanding each token encountered. Template Makefile Tokens Expanded by make_rtw lists the tokens and their expansions.

These tokens are used in several ways by the expanded makefile:

- To control the conditional behavior in the makefile. The conditionals are used to control the source file lists, library names, target to be built, and other build-related information.
- To provide the macro definitions for compiling the files, for example, `-DINTEGER_CODE=1`.

Template Makefile Tokens Expanded by make_rtw

Token	Expansion
General purpose	
`|>ADDITIONAL_LDFLAGS<|`	Linker flags automatically added by blocks.
`|>ALT_MATLAB_BIN<|`	Alternate full pathname for the MATLAB executable; value is different than value for `MATLAB_BIN` token when the full pathname contains spaces.
`|>ALT_MATLAB_ROOT<|`	Alternate full pathname for the MATLAB installation; value is different than value for `MATLAB_ROOT` token when the full pathname contains spaces.
`|>BUILDARGS<|`	Options passed to `make_rtw`. This token is provided so that the contents of your *`model`*`.mk` file changes when you change the build arguments, thus forcing an update of modules when your build options change.
`|>COMBINE_OUTPUT_UPDATE_FCNS<|`	True (1) when **Single output/update function** is selected, otherwise False (0). Used for the macro definition `-DONESTEPFCN=1`.
`|>COMPUTER<|`	Computer type. See the MATLAB `computer` command.
`|>EXPAND_LIBRARY_LOCATION<|`	Location of precompiled library file. The `TargetPreCompLibLocation` configuration parameter can override this setting. For examples, see "Control Library Location and Naming During Build" (Simulink Coder).
`|>EXPAND_LIBRARY_NAME<|`	Library name. For examples, see "Control Library Location and Naming During Build" (Simulink Coder) and "Modify the Template Makefile for rtwmakecfg" (Simulink Coder).

Token	Expansion
`|>EXPAND_LIBRARY_SUFFIX<|`	Library suffix. The `TargetLibSuffix` configuration parameter can override this setting. For examples, see "Control Library Location and Naming During Build" (Simulink Coder).
`|>EXT_MODE<|`	True (1) to enable generation of external mode support code, otherwise False (0).
`|>EXTMODE_TRANSPORT<|`	Index of transport mechanism (for example, `tcpip`, `serial`) for external mode.
`|>EXTMODE_STATIC<|`	True (1) if static memory allocation is selected for external mode. False (0) if dynamic memory allocation is selected.
`|>EXTMODE_STATIC_SIZE<|`	Size of static memory allocation buffer (if any) for external mode.
`|>GENERATE_ERT_S_FUNCTION<|`	True (1) when **Create SIL block** is selected, otherwise False (0). Used for control of the makefile target of the build.
`|>INCLUDE_MDL_TERMINATE_FCN<|`	True (1) when **Terminate function required** is selected, otherwise False (0). Used for the macro definition `-DTERMFCN==1`.
`|>INTEGER_CODE<|`	True (1) when **Support floating-point numbers** is not selected, otherwise False (0). `INTEGER_CODE` is a required macro definition when compiling the source code and is used when selecting precompiled libraries to link against.
`|>MAKEFILE_NAME<|`	*`model`*`.mk` — The name of the makefile that was created from the TMF.
`|>MAT_FILE<|`	True (1) when **MAT-file logging** is selected, otherwise False (0). `MAT_FILE` is a required macro definition when compiling the source code and also is used to include logging code in the build process.
`|>MATLAB_BIN<|`	Location of the MATLAB executable.
`|>MATLAB_ROOT<|`	Path to where MATLAB is installed.

Token	Expansion
`|>MEM_ALLOC<|`	Either `RT_MALLOC` or `RT_STATIC`. Indicates how memory is to be allocated.
`|>MEXEXT<|`	MEX-file extension. See the MATLAB `mexext` command.
`|>MODEL_MODULES<|`	Additional generated source modules. For example, you can split a large model into two files, *`model`*`.c` and *`model`*`1.c`. In this case, this token expands to *`model`*`1.c`.
`|>MODEL_MODULES_OBJ<|`	Object filenames (`.obj`) corresponding to additional generated source modules.
`|>MODEL_NAME<|`	Name of the Simulink block diagram currently being built.
`|>MULTITASKING<|`	True (1) if solver mode is multitasking, otherwise False (0).
`|>NCSTATES<|`	Number of continuous states.
`|>NUMST<|`	Number of sample times in the model.
`|>PORTABLE_WORDSIZES<|`	True (1) when **Enable portable word sizes** is selected, otherwise False (0).
`|>RELEASE_VERSION<|`	The MATLAB release version.
`|>S_FUNCTIONS<|`	List of noninlined S-function sources.
`|>S_FUNCTIONS_LIB<|`	List of S-function libraries available for linking.
`|>S_FUNCTIONS_OBJ<|`	Object (`.obj`) file list corresponding to noninlined S-function sources.
`|>SOLVER<|`	Solver source filename, for example, `ode3.c`.
`|>SOLVER_OBJ<|`	Solver object (`.obj`) filename, for example, `ode3.obj`.
`|>TARGET_LANG_EXT<|`	`c` when the **Language** selection is C, `cpp` when the **Language** selection is C++. Used in the makefile to control the extension on generated source files.
`|>TGT_FCN_LIB<|`	Specifies compiler command line options. The line in the makefile is `TGT_FCN_LIB = |>TGT_FCN_LIB<|`. Use this token in a makefile conditional statement to

<table>
<tr><th>Token</th><th>Expansion</th></tr>
<tr><td></td><td>specify a standard math library as a compiler option. Possible |>TGT_FCN_LIB<| token values are:

<table><tr><th>Value</th><th>Generates Calls To</th></tr><tr><td>Name of custom CRL</td><td>ISO®/IEC 9899:1990 C (<code>ANSI_C</code>) standard math library</td></tr><tr><td><code>ISO_C</code></td><td>ISO/IEC 9899:1999 C standard math library</td></tr><tr><td><code>ISO_C++</code></td><td>ISO/IEC 14882:2003 C++ standard math library</td></tr><tr><td><code>GNU</code></td><td>GNU extensions to the ISO/IEC 9899:1999 C standard math library</td></tr></table></td></tr>
<tr><td><code>|>TID01EQ<|</code></td><td>True (1) if sampling rates of the continuous task and the first discrete task are equal, otherwise False (0).</td></tr>
<tr><td colspan="2">S-function and build information support

Note: For examples of the tokens in this section, see “Modify the Template Makefile for rtwmakecfg” (Simulink Coder).</td></tr>
<tr><td><code>|>START_EXPAND_INCLUDES<|</code>
<code>|>EXPAND_DIR_NAME<|</code>
<code>|>END_EXPAND_INCLUDES<|</code></td><td>List of folder names to add to the include path. Additionally, the <code>ADD_INCLUDES</code> macro must be added to the <code>INCLUDES</code> line.</td></tr>
<tr><td><code>|>START_EXPAND_LIBRARIES<|</code>
<code>|>EXPAND_LIBRARY_NAME<|</code>
<code>|>END_EXPAND_LIBRARIES<|</code></td><td>List of library names.</td></tr>
<tr><td><code>|>START_EXPAND_MODULES<|</code>
<code>|>EXPAND_MODULE_NAME<|</code>
<code>|>END_EXPAND_MODULES<|</code></td><td>Library module names within <code>|>START_EXPAND_LIBRARIES<|</code> and <code>|>START_PRECOMP_LIBRARIES<|</code> library lists.</td></tr>
<tr><td><code>|>START_EXPAND_RULES<|</code>
<code>|>EXPAND_DIR_NAME<|</code>
<code>|>END_EXPAND_RULES<|</code></td><td>Makefile rules.</td></tr>
<tr><td><code>|>START_PRECOMP_LIBRARIES<|</code></td><td>List of precompiled library names.</td></tr>
</table>

Token	Expansion
`|>EXPAND_LIBRARY_NAME<|` `|>END_PRECOMP_LIBRARIES<|`	
Model reference support **Note:** For examples of the tokens in this section, see "Providing Model Referencing Support in the TMF" on page 71-85.	
`|>MASTER_ANCHOR_DIR<|`	For parallel builds, current work folder (`pwd`) at the time the build started.
`|>MODELLIB<|`	Name of the library file generated for the current model.
`|>MODELREFS<|`	List of models referenced by the top model.
`|>MODELREF_LINK_LIBS<|`	List of referenced model libraries against which the top model links.
`|>MODELREF_LINK_RSPFILE_NAME<|`	Name of a response file against which the top model links. This token is valid only for build environments that support linker response files. For an example of its use, see *`matlabroot`*`/rtw/c/grt/grt_vc.tmf`.
`|>MODELREF_TARGET_TYPE<|`	Type of target being built. Possible values are • `NONE`: Standalone model or top model referencing other models • `RTW`: Model reference target build • `SIM`: Model reference simulation target build
`|>RELATIVE_PATH_TO_ANCHOR<|`	Relative path, from the location of the generated makefile, to the MATLAB working folder.
`|>START_DIR<|`	Current work folder (`pwd`) at the time the build started. This token is required for parallel builds.

`|`
`>START_MDLREFINC_EXPAND_INCLUDES<`
`|>MODELREF_INC_PATH<|`
`|>END_MDLREFINC_EXPAND_INCLUDES<|`
	List of include paths for models referenced by the top model.

`|>SHARED_BIN_DIR<|`	Folder for the library file built from the shared source files.

Token	Expansion
`|>SHARED_LIB<|`	Library file built from the shared source files, including the path to the library folder.
`|>SHARED_SRC<|`	Shared source files specification, including the path to the shared utilities folder.
`|>SHARED_SRC_DIR<|`	Folder for shared source files.

These tokens are expanded by substitution of parameter values known to the build process. For example, if the source model contains blocks with two different sample times, the TMF statement

```
NUMST = |>NUMST<|
```

expands to the following in *`model`*`.mk`.

```
NUMST = 2
```

In addition to the above, `make_rtw` expands tokens from other sources:

- Target-specific tokens defined in the target options of the Configuration Parameters dialog box
- Structures in the `rtwoptions` section of the system target file. Structures in the `rtwoptions` structure array that contain the field `makevariable` are expanded.

The following example is extracted from *`matlabroot`*`/rtw/c/grt/grt.tlc`. The section starting with `BEGIN_RTW_OPTIONS` contains MATLAB code that sets up `rtwoptions`. The following directive causes the `|>EXT_MODE<|` token to be expanded to `1` (on) or `0` (off), depending on how you set the external mode options.

```
rtwoptions(2).makevariable = 'EXT_MODE'
```

Invoke the make Utility

- “make Command” on page 71-68
- “make Utility Versions” on page 71-69

make Command

After creating *`model`*`.mk` from your TMF, the build process invokes a `make` command. To invoke `make`, the build process issues this command.

```
makecommand -f model.mk
```

makecommand is defined by the MAKECMD macro in your target's TMF (see "Structure of the Template Makefile" on page 71-69). You can specify additional options to make in the **Make command** field of the **Code Generation** pane. (See the sections "Specify a Make Command" (Simulink Coder) and "Template Makefiles and Make Options" (Simulink Coder).)

For example, specifying OPT_OPTS=-O2 in the **Make command** field causes make_rtw to generate the following make command.

```
makecommand -f model.mk OPT_OPTS=-O2
```

A comment at the top of the TMF specifies the available make command options. If these options do not provide you with enough flexibility, you can configure your own TMF.

make Utility Versions

The make utility lets you control nearly every aspect of building your real-time program. There are several different versions of make available. The code generator provides the Free Software Foundation GNU make for both UNIX[11] and PC platforms in platform-specific subfolders under

```
matlabroot/bin
```

It is possible to use other versions of make with the code generator, although GNU Make is recommended. To be compatible with the code generator, verify that your version of make supports the following command format.

```
makecommand -f model.mk
```

Structure of the Template Makefile

A TMF has multiple sections, including the following:

- Abstract — Describes what the makefile targets. Here is a representative abstract from the GRT TMFs in *matlabroot*/rtw/c/grt (open):

```
# File    : grt_lcc.tmf
#
# Abstract:
#       Template makefile for building a PC-based stand-alone generic real-time
#       version of Simulink model using generated C code and LCC compiler
#       Version 2.4.
#
```

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

```
#       This makefile attempts to conform to the guidelines specified in the
#       IEEE Std 1003.2-1992 (POSIX) standard. It is designed to be used
#       with GNU Make (gmake) which is located in matlabroot/bin/win32.
#
#       Note that this template is automatically customized by the build
#       procedure to create "<model>.mk"
#
#       The following defines can be used to modify the behavior of the
#       build:
#         OPT_OPTS       - Optimization options. Default is none. To enable
#                          debugging specify as OPT_OPTS=-g4.
#         OPTS           - User specific compile options.
#         USER_SRCS      - Additional user sources, such as files needed by
#                          S-functions.
#         USER_INCLUDES  - Additional include paths
#                          (i.e. USER_INCLUDES="-Iwhere-ever -Iwhere-ever2")
#                          (For Lcc, have a '/'as file seperator before the
#                          file name instead of a '\' .
#                          i.e.,  d:\work\proj1/myfile.c - reqd for 'gmake')
#
#       This template makefile is designed to be used with a system target
#       file that contains 'rtwgensettings.BuildDirSuffix'. See grt.tlc.
```

- `Macros read by make_rtw` section — Defines macros that tell `make_rtw` how to process the TMF. Here is a representative `Macros read by make_rtw` section from the GRT TMFs in *`matlabroot`*`/rtw/c/grt` (open):

```
#----------------------- Macros read by make_rtw ------------------------------
#
# The following macros are read by the build procedure:
#
#  MAKECMD         - The command that invokes the make utility
#  HOST            - The platform (for example, PC) for which this TMF is written
#  SHELL           - An operating system shell command (for example, cmd) for the platform
#  BUILD           - The flag that indicates whether to invoke make from the build procedure
#  SYS_TARGET_FILE - Name of system target file.

MAKECMD         = "%MATLAB%\bin\win32\gmake"
HOST            = PC
SHELL           = cmd
BUILD           = yes
SYS_TARGET_FILE = grt.tlc
BUILD_SUCCESS   = *** Created
COMPILER_TOOL_CHAIN = lcc

MAKEFILE_FILESEP = /
```

  The macros in this section might include:

  - `MAKECMD` — Specifies the command used to invoke the make utility. For example, if `MAKECMD = mymake`, then the `make` command invoked is

    ```
    mymake -f model.mk
    ```

- HOST — Specifies the platform targeted by this TMF. This can be PC, UNIX, *computer_name* (see the MATLAB computer command), or ANY.
- SHELL — Specifies an operating system shell command for the platform. For Windows, this can be cmd.
- BUILD — Instructs make_rtw whether or not it should invoke make from the build procedure. Specify yes or no.
- SYS_TARGET_FILE — Specifies the name of the system target file or the value any. This is used for consistency checking by make_rtw to verify the system target file specified in the **Target selection** panel of the **Code Generation** pane of the Configuration Parameters dialog box. If you specify any, you can use the TMF with any system target file.
- BUILD_SUCCESS — Optional macro that specifies the build success message to be displayed for make completion on the PC. For example,

  ```
  BUILD_SUCCESS = ### Successful creation of
  ```

  The BUILD_SUCCESS macro, if used, replaces the standard build success message found in the TMFs distributed with the bundled code generator targets (such as GRT):

  ```
  @echo ### Created executable $(MODEL).exe
  ```

  Your TMF must include either the standard build success message, or use the BUILD_SUCCESS macro. For an example of the use of BUILD_SUCCESS, see *matlabroot*/rtw/c/grt/grt_lcc.tmf or the code example above this list of macros.
- BUILD_ERROR — Optional macro that specifies the build error message to be displayed when an error is encountered during the make procedure. For example,

  ```
  BUILD_ERROR = ['Error while building ', modelName]
  ```
- VERBOSE_BUILD_OFF_TREATMENT = PRINT_OUTPUT_ALWAYS — Optional macro to include if you want the makefile output to be displayed regardless of the setting of the **Verbose build** option in the **Code Generation** > **Debug** pane.
- COMPILER_TOOL_CHAIN — For builds on Windows systems, specifies which compiler setup file — located in *matlabroot*/toolbox/rtw/rtw (open) — to use:
  - lcc selects setup_for_lcc.m
  - vc selects setup_for_visual.m

- `vcx64` selects `setup_for_visual_x64.m`
- `default` selects `setup_for_default.m`

For builds on UNIX systems, specify `unix`. Other values are flagged as unknown and `make_rtw` uses `setup_for_default.m`.

Note: Do not omit `COMPILER_TOOL_CHAIN` or leave it unspecified in your TMF. If your compiler is not the host compiler, specify `COMPILER_TOOL_CHAIN = default`.

- `DOWNLOAD` — An optional macro that you can specify as yes or no. If specified as yes (and `BUILD=yes`), then `make` is invoked a second time with the download target.

  ```
  make -f model.mk download
  ```

- `DOWNLOAD_SUCCESS` — An optional macro that you can use to specify the download success message to be used when looking for a completed download. For example,

  ```
  DOWNLOAD_SUCCESS = ### Downloaded
  ```

- `DOWNLOAD_ERROR` — An optional macro that you can use to specify the download error message to be displayed when an error is encountered during the download. For example,

  ```
  DOWNLOAD_ERROR = ['Error while downloading ', modelName]
  ```

- `Tokens expanded by make_rtw` section — Defines the tokens that `make_rtw` expands. Here is a brief excerpt from a representative `Tokens expanded by make_rtw` section from the GRT TMFs in *`matlabroot`*`/rtw/c/grt` (open):

```
#---------------------- Tokens expanded by make_rtw ----------------------------
#
# The following tokens, when wrapped with "|>" and "<|" are expanded by the
# build procedure.
#
#  MODEL_NAME          - Name of the Simulink block diagram
#  MODEL_MODULES       - Any additional generated source modules
#  MAKEFILE_NAME       - Name of makefile created from template makefile <model>.mk
#  MATLAB_ROOT         - Path to where MATLAB is installed.
...

MODEL                = |>MODEL_NAME<|
MODULES              = |>MODEL_MODULES<|
MAKEFILE             = |>MAKEFILE_NAME<|
```

```
MATLAB_ROOT          = |>MATLAB_ROOT<|
...
```

For more information about TMF tokens, see Template Makefile Tokens Expanded by make_rtw.

- Subsequent sections vary based on compiler, host, and target. Some common sections include `Model and reference models`, `External mode`, `Tool Specifications` or `Tool Definitions`, `Include Path`, `C Flags`, `Additional Libraries`, and `Source Files`.
- `Rules` section — Contains the make rules used in building an executable from the generated source code. The build rules are typically specific to your version of make. The `Rules` section might be followed by related sections such as `Dependencies`.

Customize and Create Template Makefiles

Introduction

This section describes the mechanics of setting up a custom template makefile (TMF) and incorporating it into the build process. It also discusses techniques for modifying a TMF and MATLAB file mechanisms associated with the TMF.

Before creating a custom TMF, you should read "Folder and File Naming Conventions" on page 71-11 to understand the folder structure and MATLAB path requirements for custom targets.

Setting Up a Template Makefile

To customize or create a new TMF, you should copy an existing GRT or ERT TMF from one of the following locations:

```
matlabroot/rtw/c/grt (open)
matlabroot/rtw/c/ert (open)
```

Place the copy in the same folder as the associated system target file (STF). Usually, this is the `mytarget/mytarget` folder within the target folder structure. Then, rename your TMF (for example, `mytarget.tmf`) and modify it.

To allow the build process to locate and select your TMF, you must provide information in the STF file header (see "System Target File Structure" on page 71-30). For a target that implements a single TMF, the standard way to specify the TMF to be used in the build process is to use the TMF directive of the STF file header.

```
TMF: mytarget.tmf
```

Using Macros and Pattern Matching Expressions in a Template Makefile

This section shows, through an example, how to use macros and file-pattern-matching expressions in a TMF to generate commands in the *`model`*`.mk` file.

The make utility processes the *`model`*`.mk` makefile and generates a set of commands based upon dependency rules defined in *`model`*`.mk`. After `make` generates the set of commands for building or rebuilding `test`, `make` executes them.

For example, to build a program called `test`, `make` must link the object files. However, if the object files don't exist or are out of date, `make` must compile the source code. Thus there is a dependency between source and object files.

Each version of `make` differs slightly in its features and how rules are defined. For example, consider a program called `test` that gets created from two sources, `file1.c` and `file2.c`. Using most versions of `make`, the dependency rules would be

```
test: file1.o file2.o
        cc -o test file1.o file2.o

file1.o: file1.c
        cc -c file1.c

file2.o: file2.c
        cc -c file2.c
```

In this example, a UNIX[12] environment is assumed. In a PC environment the file extensions and compile and link commands are different.

In processing the first rule

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

```
test: file1.o file2.o
```

`make` sees that to build `test`, it needs to build `file1.o` and `file2.o`. To build `file1.o`, `make` processes the rule

```
file1.o: file1.c
```

If `file1.o` doesn't exist, or if `file1.o` is older than `file1.c`, `make` compiles `file1.c`.

The format of TMFs follows the above example. Our TMFs use additional features of `make` such as macros and file-pattern-matching expressions. In most versions of `make`, a macro is defined with

```
MACRO_NAME = value
```

References to macros are made with `$(MACRO_NAME)`. When `make` sees this form of expression, it substitutes *`value`* for `$(MACRO_NAME)`.

You can use pattern matching expressions to make the dependency rules more general. For example, using GNU[13] Make, you could replace the two "`file1.o: file1.c`" and "`file2.o: file2.c`" rules with the single rule

```
%.o : %.c
        cc -c $<
```

Note that `$<` in the previous example is a special macro that equates to the dependency file (that is, `file1.c` or `file2.c`). Thus, using macros and the "`%`" pattern matching character, the previous example can be reduced to

```
SRCS = file1.c file2.c
OBJS = $(SRCS:.c=.o)

test: $(OBJS)
        cc -o $@ $(OBJS)

%.o : %.c
        cc -c $<
```

Note that the `$@` macro above is another special macro that equates to the name of the current dependency target, in this case `test`.

This example generates the list of objects (`OBJS`) from the list of sources (`SRCS`) by using the text substitution feature for macro expansion. It replaces the source file extension

13. GNU is a registered trademark of the Free Software Foundation.

(for example, `.c`) with the object file extension (`.o`). This example also generalized the build rule for the program, `test`, to use the special `"$@"` macro.

Customizing Generated Makefiles with rtwmakecfg

TMFs provide tokens that let you add the following items to generated makefiles:

- Source folders
- Include folders
- Run-time library names
- Run-time module objects

S-functions can add this information to the makefile by using an `rtwmakecfg.m` file function. This function is particularly useful when building a model that contains one or more of your S-Function blocks, such as device driver blocks.

To add information pertaining to an S-function to the makefile,

1. Create the function `rtwmakecfg` in file `rtwmakecfg.m`. The code generator associates this file with your S-function based on its folder location.
2. Modify your target's TMF such that it supports macro expansion for the information returned by `rtwmakecfg` functions.

After the TLC phase of the build process, when generating a makefile from the TMF, the build process searches for an `rtwmakecfg.m` file in the folder that contains the S-function component. If it finds the file, the build process calls the `rtwmakecfg` function. For more information, see “Use rtwmakecfg.m API to Customize Generated Makefiles” (Simulink Coder).

Supporting Continuous Time in Custom Targets

If you want your custom ERT-based target to support continuous time, you must update your template makefile (TMF) and the static main program module (for example, `mytarget_main.c`) for your target.

Template Makefile Modifications

Add the `NCSTATES` token expansion after the `NUMST` token expansion, as follows:

```
NUMST = |>NUMST<|
NCSTATES = |>NCSTATES<|
```

In addition, add NCSTATES to the CPP_REQ_DEFINES macro, as in the following example:

```
CPP_REQ_DEFINES = -DMODEL=$(MODEL) -DNUMST=$(NUMST) -DNCSTATES=$(NCSTATES) \
-DMAT_FILE=$(MAT_FILE)
-DINTEGER_CODE=$(INTEGER_CODE) \
-DONESTEPFCN=$(ONESTEPFCN) -DTERMFCN=$(TERMFCN) \
-DHAVESTDIO
-DMULTI_INSTANCE_CODE=$(MULTI_INSTANCE_CODE) \
```

Modifications to Main Program Module

The main program module defines a static main function that manages task scheduling for the supported tasking modes of single- and multiple-rate models. NUMST (the number of sample times in the model) determines whether the main function calls multirate or single-rate code. However, when a model uses continuous time, do not rely on NUMST directly.

When the model has continuous time and the flag TID01EQ is true, both continuous time and the fastest discrete time are treated as one rate in generated code. The code associated with the fastest discrete rate is guarded by a major time step check. When the model has only two rates, and TID01EQ is true, the generated code has a single-rate call interface.

To support models that have continuous time, update the static main module to take TID01EQ into account, as follows:

1. Before NUMST is referenced in the file, add the following code:

   ```
   #if defined(TID01EQ) && TID01EQ == 1 && NCSTATES == 0
   #define DISC_NUMST (NUMST - 1)
   #else
   #define DISC_NUMST NUMST
   #endif
   ```

2. Replace instances of NUMST in the file by DISC_NUMST.

Model Reference Considerations

See “Support Model Referencing” on page 71-83 for important information on TMF modifications you may need to make to support the code generator model referencing features.

Note: If you are using a TMF without the variables SHARED_SRC or MODELREFS, the file might have been used with a previous release of Simulink software. If you want your

TMF to support model referencing, add either variable `SHARED_SRC` or `MODELREFS` to the make file.

More About

- "About Embedded Target Development" (Simulink Coder)
- "Sample Custom Targets" (Simulink Coder)
- "Customize System Target Files" (Simulink Coder)
- "Custom Target Optional Features" (Simulink Coder)
- "Support Toolchain Approach with Custom Target" (Simulink Coder)

Custom Target Optional Features

This section describes how to configure a custom embedded target to support these optional features.

To ...	Use Target Configuration Parameters ...	For more information, see ...
Indicate a custom target is toolchain-compliant	`UseToolchainInfoCompliaGenerateMakefile`	"Support Toolchain Approach with Custom Target" (Simulink Coder)
Build a model that includes referenced models and uses a custom target	`ModelReferenceCompliant` `ParMdlRefBuildCompliant` (parallel build support)	"Support Model Referencing" (Simulink Coder)
Control the compiler optimization level building generated code for a custom target	`CompOptLevelCompliant`	"Support Compiler Optimization Level Control" (Simulink Coder)
Control C function prototypes of initialize and step functions that are generated for a model that uses a custom target	`ModelStepFunctionProtot Compliant` (ERT only)	"Support C Function Prototype Control" (Simulink Coder)
Control C++ class interfaces that are generated for a model that uses a custom target	`CPPClassGenCompliant` (ERT only)	"Support C++ Class Interface Control" (Simulink Coder)
Enable concurrent execution of multiple tasks on a multicore platform for a model that uses a custom target	`ConcurrentExecutionComp`	"Support Concurrent Execution of Multiple Tasks" (Simulink Coder)

The required configuration changes are modifications to your system target file (STF), and in some cases also modifications to your template makefile (TMF) or your custom static main program.

The API for STF callbacks provides a function `SelectCallback` for use in STFs. `SelectCallback` is associated with the target rather than with its individual options. If you implement a `SelectCallback` function for a target, it is triggered whenever the user selects the target in the System Target File Browser.

The API provides the functions `slConfigUIGetVal`, `slConfigUISetEnabled`, and `slConfigUISetVal` for controlling custom target configuration options from a user-written `SelectCallback` function. (For function descriptions and examples, see the function reference pages.)

The general requirements for supporting one of the optional features include:

- To support model referencing or compiler optimization level control, the target must be derived from the GRT or the ERT target. To support C function prototype control or C++ class interface control, the target must be derived from the ERT target.
- The system target file (STF) must declare feature compliance by including one of the target configuration parameters listed above in a `SelectCallback` function call.
- Additional changes such as TMF modifications or static main program modifications may be required, depending on the feature. See the detailed steps in the subsections for individual features.

For an example that shows how to configure custom target optional features, see "Customize System Target Files" (Simulink Coder).

More About

- "About Embedded Target Development" (Simulink Coder)
- "Sample Custom Targets" (Simulink Coder)
- "Support Toolchain Approach with Custom Target" (Simulink Coder)
- "Support Model Referencing" (Simulink Coder)
- "Support Compiler Optimization Level Control" (Simulink Coder)
- "Support C Function Prototype Control" (Simulink Coder)
- "Support C++ Class Interface Control" (Simulink Coder)
- "Support Concurrent Execution of Multiple Tasks" (Simulink Coder)

Support Toolchain Approach with Custom Target

This section describes how to configure a custom system target file to support builds with the toolchain approach.

In the Configuration Parameters dialog box, on the Code Generation pane of, you can present either the build controls for the toolchain approach or the template makefile approach. The model parameters that contribute to determining which build controls appear include these parameters.

Model Parameter	Value	Notes
`UseToolchainInfoCompliant`	`on`	For toolchain approach, set this parameter to `'on'`. For TMF approach, set this parameter to `'off'`.
`GenerateMakefile`	`on`	For toolchain approach, set this parameter to `'on'`.

When the dialog box detects that the selected target has these properties, the dialog box recognizes the target as toolchain-compliant and displays the build controls for the toolchain approach.

Because the custom target file cannot set these properties directly, use a `SelectCallback` function in the custom target file to set the properties. The `SelectCallback` function call in the RTW_OPTION section of the TLC file can take the form:

```
rtwgensettings.SelectCallback = 'enableToolchainCompliant(hSrc, hDlg)';
```

A corresponding callback function can contain:

```
function enableToolchainCompliant(hSrc, hDlg)
  hCS = hSrc.getConfigSet();

  % The following parameters enable toolchain compliance.
  slConfigUISetVal(hDlg, hSrc, 'UseToolchainInfoCompliant', 'on');
  hCS.setProp('GenerateMakefile','on');

  % The following parameters are not required for toolchain compliance.
  % But, it is recommended practice to set these default values and
  % disable the parameters (as shown).
  hCS.setProp('RTWCompilerOptimization','off');
  hCS.setProp('MakeCommand','make_rtw');
  hCS.setPropEnabled('RTWCompilerOptimization',false);
```

```
  hCS.setPropEnabled('MakeCommand',false);
end
```

When you select your custom target, the configuration parameters dialog box displays the toolchain approach build controls. For an example, see “Create a Custom Target Configuration” on page 71-48.

For an example that shows how to configure custom target optional features, see “Customize System Target Files” (Simulink Coder).

More About

- “Customize System Target Files” (Simulink Coder)
- “Support Model Referencing” (Simulink Coder)
- “Support Compiler Optimization Level Control” (Simulink Coder)
- “Support C Function Prototype Control” (Simulink Coder)
- “Support C++ Class Interface Control” (Simulink Coder)
- “Support Concurrent Execution of Multiple Tasks” (Simulink Coder)

Support Model Referencing

This section describes how to configure a custom embedded target to support model referencing. Without the described modifications, you will not be able to use the custom target when building a model that includes referenced models. If you do not intend to use referenced models with your target, you can skip this section. If you later find that you need to use referenced models, you can upgrade your target then.

In this section...

About Model Referencing with a Custom Target

The requirements for supporting model referencing are as follows:

- The target must be derived from the GRT target or the ERT target.
- The system target file (STF) must declare model reference compliance, as described in "Declaring Model Referencing Compliance" on page 71-84.
- The template makefile (TMF) must define some entities that support model referencing, as described in "Providing Model Referencing Support in the TMF" on page 71-85.
- The TMF must support using the Shared Utilities folder, as described in "Supporting the Shared Utilities Folder" on page 71-88.

Optionally, you can provide additional capabilities that support model referencing:

- You can configure a target to support parallel builds for large model reference hierarchies (see "Reduce Build Time for Referenced Models" (Simulink Coder)). To do

this, you must modify the STF and TMF for parallel builds as described in "Declaring Model Referencing Compliance" on page 71-84 and "Providing Model Referencing Support in the TMF" on page 71-85.

- If your target supports parallel builds for large model reference hierarchies, you can additionally set up automatic verification of MATLAB Distributed Computing Server workers, as described in "Verifying Worker Configuration for Parallel Builds of Model Reference Hierarchies (Optional)" on page 71-92.
- You can modify hook files to handle referenced models differently than top models to prevent resource conflicts, as described in "Preventing Resource Conflicts (Optional)" on page 71-94.

See "Overview of Model Referencing" (Simulink) for information about model referencing in Simulink models, and "Generate Code for Referenced Models" (Simulink Coder) for information about model referencing in generated code.

For an example that shows how to configure custom target optional features, see "Customize System Target Files" (Simulink Coder).

Declaring Model Referencing Compliance

To declare model reference compliance for your target, you must implement a callback function that sets the `ModelReferenceCompliant` flag, and then install the callback function in the `SelectCallback` field of the `rtwgensettings` structure in your STF. The callback function is triggered whenever the user selects the target in the System Target File Browser. For example, the following STF code installs a `SelectCallback` function named `custom_select_callback_handler`:

```
rtwgensettings.SelectCallback = 'custom_select_callback_handler(hDlg,hSrc)';
```

The arguments to the `SelectCallback` function (`hDlg`, `hSrc`) are handles to private data used by the callback API functions. These handles are restricted to use in STF callback functions. They should be passed in without alteration.

Your callback function should set the `ModelReferenceCompliant` flag as follows:

```
slConfigUISetVal(hDlg,hSrc,'ModelReferenceCompliant','on');
slConfigUISetEnabled(hDlg,hSrc,'ModelReferenceCompliant',false);
```

If you might use the target to build models containing large model reference hierarchies, consider configuring the target to support parallel builds, as discussed in "Reduce Build Time for Referenced Models" (Simulink Coder).

To configure a target for parallel builds, your callback function must also set the `ParMdlRefBuildCompliant` flag as follows:

```
slConfigUISetVal(hDlg,hSrc,'ParMdlRefBuildCompliant','on');
slConfigUISetEnabled(hDlg,hSrc,'ParMdlRefBuildCompliant',false);
```

For more information about the STF callback API, see the `slConfigUIGetVal`, `slConfigUISetEnabled`, and `slConfigUISetVal` function reference pages.

Providing Model Referencing Support in the TMF

Do the following to configure the template makefile (TMF) to support model referencing:

1 Add the following make variables and tokens to be expanded when the makefile is generated:

```
MODELREFS                  = |>MODELREFS<|
MODELLIB                   = |>MODELLIB<|
MODELREF_LINK_LIBS         = |>MODELREF_LINK_LIBS<|
MODELREF_LINK_RSPFILE      = |>MODELREF_LINK_RSPFILE_NAME<|
MODELREF_INC_PATH         = |>START_MDLREFINC_EXPAND_INCLUDES<|\
    -I|>MODELREF_INC_PATH<| |>END_MDLREFINC_EXPAND_INCLUDES<|
RELATIVE_PATH_TO_ANCHOR    = |>RELATIVE_PATH_TO_ANCHOR<|
MODELREF_TARGET_TYPE       = |>MODELREF_TARGET_TYPE<|
```

The following code excerpt shows how makefile tokens are expanded for a referenced model.

```
MODELREFS                =
MODELLIB                 = engine3200cc_rtwlib.a
MODELREF_LINK_LIBS       =
MODELREF_LINK_RSPFILE    =
MODELREF_INC_PATH        =
RELATIVE_PATH_TO_ANCHOR  = ../../..
MODELREF_TARGET_TYPE     = RTW
```

The following code excerpt shows how makefile tokens are expanded for the top model that references the referenced model.

```
MODELREFS                = engine3200cc transmission
MODELLIB                 = archlib.a
MODELREF_LINK_LIBS       = engine3200cc_rtwlib.a transmission_rtwlib.a
MODELREF_LINK_RSPFILE    =
MODELREF_INC_PATH      = -I../slprj/ert/engine3200cc -I../slprj/ert/transmission
RELATIVE_PATH_TO_ANCHOR  = ..
MODELREF_TARGET_TYPE     = NONE
```

Token	Expands to
`MODELREFS` for the top model	List of referenced model names.

Token	Expands to
MODELLIB	Name of the library generated for the model.
MODELREF_LINK_LIBS token for the top model	List of referenced model libraries that the top model links against.
MODELREF_LINK_RSPFILE token for the top model	Name of a response file that the top model links against. This token is valid only for build environments that support linker response files. For an example of its use, see *matlabroot*/rtw/c/grt/grt_vc.tmf.
MODELREF_INC_PATH token for the top model	Include path to the referenced models.
RELATIVE_PATH_TO_ANCHOR	Relative path, from the location of the generated makefile, to the MATLAB working folder.
MODELREF_TARGET_TYPE	Signifies the type of target being built. Possible values are • NONE: Standalone model or top model referencing other models • RTW: Model reference target build • SIM: Model reference simulation target build

If you are configuring your target to support parallel builds, as discussed in "Reduce Build Time for Referenced Models" (Simulink Coder), you must also add the following token definitions to your TMF:

```
START_DIR = |>START_DIR<|
MASTER_ANCHOR_DIR = |>MASTER_ANCHOR_DIR<|
```

Token	Expands to
START_DIR	Current work folder (pwd) at the time the build started.
MASTER_ANCHOR_DIR	Current work folder (pwd) at the time the build started.

2. Add `RELATIVE_PATH_TO_ANCHOR` and `MODELREF_INC_PATH` include paths to the overall `INCLUDES` variable.

```
INCLUDES = -I. -I$(RELATIVE_PATH_TO_ANCHOR) $(MATLAB_INCLUDES) $(ADD_INCLUDES) \
    $(USER_INCLUDES) $(MODELREF_INC_PATH) $(SHARED_INCLUDES)
```

3. Change the `SRCS` variable in your TMF so that it initially lists only common modules. Additional modules are then appended conditionally, as described in the next step. For example, change

```
SRCS  = $(MODEL).c $(MODULES) ert_main.c $(ADD_SRCS) $(EXT_SRC)
```

to

```
SRCS = $(MODULES) $(S_FUNCTIONS)
```

4. Create variables to define the final target of the makefile. You can remove variables that may have existed for defining the final target. For example, remove

```
PROGRAM    = ../$(MODEL)
```

and replace it with

```
ifeq ($(MODELREF_TARGET_TYPE), NONE)
  # Top model for RTW
  PRODUCT            = $(RELATIVE_PATH_TO_ANCHOR)/$(MODEL)
  BIN_SETTING        = $(LD) $(LDFLAGS) -o $(PRODUCT) $(SYSLIBS)
  BUILD_PRODUCT_TYPE = "executable"
  # ERT based targets
  SRCS               += $(MODEL).c ert_main.c $(EXT_SRC)
  # GRT based targets
  # SRCS             += $(MODEL).c grt_main.c rt_sim.c $(EXT_SRC) $(SOLVER)

else
  # sub-model for RTW
  PRODUCT            = $(MODELLIB)
  BUILD_PRODUCT_TYPE = "library"
endif
```

5. Create rules for the final target of the makefile (replace existing final target rules). For example:

```
ifeq ($(MODELREF_TARGET_TYPE),NONE)
  # Top model for RTW
  $(PRODUCT) : $(OBJS) $(SHARED_LIB) $(LIBS) $(MODELREF_LINK_LIBS)
               $(BIN_SETTING) $(LINK_OBJS) $(MODELREF_LINK_LIBS)
               $(SHARED_LIB) $(LIBS)
               @echo "### Created $(BUILD_PRODUCT_TYPE): $@"
else
  # sub-model for RTW
  $(PRODUCT) : $(OBJS) $(SHARED_LIB) $(LIBS)
               @rm -f $(MODELLIB)
               $(ar) ruvs $(MODELLIB) $(LINK_OBJS)
```

```
                @echo "### Created $(MODELLIB)"
                @echo "### Created $(BUILD_PRODUCT_TYPE): $@"
endif
```

6 Create a rule to allow referenced models to compile files that reside in the MATLAB working folder (pwd).

```
%.o : $(RELATIVE_PATH_TO_ANCHOR)/%.c
 $(CC) -c $(CFLAGS) $<
```

Note: If you are using a TMF without the variables SHARED_SRC or MODELREFS, the file might have been used with a previous release of Simulink software. If you want your TMF to support model referencing, add either variable SHARED_SRC or MODELREFS to the make file.

Controlling Configuration Option Value Agreement

By default, the value of a configuration option defined in the system target file for a TLC-based custom target must be the same in any referenced model and its parent model. To relax this requirement, include the `modelReferenceParameterCheck` field in the `rtwoptions` structure element that defines the configuration option, and set the value of the field to `'off'`. For example:

```
rtwoptions(2).prompt         = 'My Custom Parameter';
rtwoptions(2).type           = 'Checkbox';
rtwoptions(2).default        = 'on';
rtwoptions(2).modelReferenceParameterCheck  = 'on';
rtwoptions(2).tlcvariable    = 'mytlcvariable';
...
```

The configuration option **My Custom Parameter** can differ in a referenced model and its parent model. See “Customize System Target Files” on page 71-29 for information about TLC-based system target files, and rtwoptions Structure Fields Summary for a list of `rtwoptions` fields.

Supporting the Shared Utilities Folder

Overview

The makefile used by the build process must support compiling and creating libraries, and so on, from the locations in which the code is generated. Therefore, you need to update your makefile and the model reference build process to support the shared utilities location. The **Shared code placement** options have the following requirements:

- `Auto`
  - Standalone model build — Build files go to the build folder; makefile is not updated.
  - Referenced model or top model build — Use shared utilities folder; makefile requires full model reference support.
- `Shared location`
  - Standalone model build — Use shared utilities folder; makefile requires shared location support.
  - Referenced model or top model build — Use shared utilities folder; makefile requires full model reference support.

The shared utilities folder (`slprj/`*`target`*`/_sharedutils`) typically stores generated utility code that is common between a top model and the models it references. You can also force the build process to use a shared utilities folder for a standalone model. See "Manage Build Process Folders" (Simulink Coder) for details.

If you want your target to support compilation of code generated in the shared utilities folder, several updates to your template makefile (TMF) are required. Support for Model Reference builds requires the shared utilities folder. See the preceding sections to learn about additional updates for supporting Model Reference builds.

The exact syntax of the changes can vary due to differences in the make utility and compiler/archiver tools used by your target. The examples below are based on the GNU[14] make utility. You can find the following updated TMF examples for GNU and Microsoft Visual C++ make utilities in the GRT and ERT target folders:

- GRT: *`matlabroot`*`/rtw/c/grt` (open)
  - `grt_lcc.tmf`
  - `grt_vc.tmf`

14. GNU is a registered trademark of the Free Software Foundation.

  - `grt_unix.tmf`
- ERT: *`matlabroot`*`/rtw/c/ert` (open)
  - `ert_lcc.tmf`
  - `ert_vc.tmf`
  - `ert_unix.tmf`

Use the GRT or ERT examples as a guide to the location, within the TMF, of the changes and additions described below.

Note The ERT-based TMFs contain extra code to handle generation of ERT S-functions and Model Reference simulation targets. Your target does not need to handle these cases.

Implementing Shared Utilities Folder Support

Make the following changes to your TMF to support the shared utilities folder:

1. Add the following make variables and tokens to be expanded when the makefile is generated:

   ```
   SHARED_SRC      = |>SHARED_SRC<|
   SHARED_SRC_DIR  = |>SHARED_SRC_DIR<|
   SHARED_BIN_DIR  = |>SHARED_BIN_DIR<|
   SHARED_LIB      = |>SHARED_LIB<|
   ```

   `SHARED_SRC` specifies the shared utilities folder location and the source files in it. A typical expansion in a makefile is

   ```
   SHARED_SRC      = ../slprj/ert/_sharedutils/*.c
   ```

   `SHARED_LIB` specifies the library file built from the shared source files, as in the following expansion.

   ```
   SHARED_LIB      = ../slprj/ert/_sharedutils/rtwshared.lib
   ```

   `SHARED_SRC_DIR` and `SHARED_BIN_DIR` allow specification of separate folders for shared source files and the library compiled from the source files. In the current release, the TMFs use the same path, as in the following expansions.

   ```
   SHARED_SRC_DIR  = ../slprj/ert/_sharedutils
   ```

```
SHARED_BIN_DIR  = ../slprj/ert/_sharedutils
```

2 Set the SHARED_INCLUDES variable according to whether shared utilities are in use. Then append it to the overall INCLUDES variable.

```
SHARED_INCLUDES =
ifneq ($(SHARED_SRC_DIR),)
SHARED_INCLUDES = -I$(SHARED_SRC_DIR)
endif

INCLUDES = -I. $(MATLAB_INCLUDES) $(ADD_INCLUDES) \
           $(USER_INCLUDES) $(SHARED_INCLUDES)
```

3 Update the SHARED_SRC variable to list shared files explicitly.

```
SHARED_SRC := $(wildcard $(SHARED_SRC))
```

4 Create a SHARED_OBJS variable based on SHARED_SRC.

```
SHARED_OBJS = $(addsuffix .o, $(basename $(SHARED_SRC)))
```

5 Create an OPTS (options) variable for compilation of shared utilities.

```
SHARED_OUTPUT_OPTS = -o $@
```

6 Provide a rule to compile the shared utility source files.

```
$(SHARED_OBJS) : $(SHARED_BIN_DIR)/%.o : $(SHARED_SRC_DIR)/%.c
 $(CC) -c $(CFLAGS) $(SHARED_OUTPUT_OPTS) $<
```

7 Provide a rule to create a library of the shared utilities. The following example is based on UNIX[15].

```
$(SHARED_LIB) : $(SHARED_OBJS)
   @echo "### Creating $@ "
   ar r $@ $(SHARED_OBJS)
   @echo "### Created  $@ "
```

Note: Depending on your make utility, you may be able to combine Steps 6 and 7 into one rule. For example, gmake (used with ert_unix.tmf) uses:

```
$(SHARED_LIB) : $(SHARED_SRC)
    @echo "### Creating $@ "
    cd $(SHARED_BIN_DIR); $(CC) -c $(CFLAGS) $(GCC_WALL_FLAG_MAX) $(notdir $?)
    ar ruvs $@ $(SHARED_OBJS)
    @echo "### $@ Created "
```

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

See this and other examples in the files `ert_vc.tmf`, `ert_lcc.tmf`, and `ert_unix.tmf` located at *`matlabroot`*`/rtw/c/ert` (open).

8 Add `SHARED_LIB` to the rule that creates the final executable.

```
$(PROGRAM) : $(OBJS) $(LIBS) $(SHARED_LIB)
 $(LD) $(LDFLAGS) -o $@ $(LINK_OBJS) $(LIBS)
$(SHARED_LIB) $(SYSLIBS)
 @echo "### Created executable: $(MODEL)"
```

9 Remove explicit references to `rt_nonfinite.c` from your TMF. For example, change

```
ADD_SRCS = $(RTWLOG) rt_nonfinite.c
```

to

```
ADD_SRCS = $(RTWLOG)
```

Note If your target interfaces to a development environment that is not makefile based, you must make equivalent changes to provide information to your target compilation environment.

Verifying Worker Configuration for Parallel Builds of Model Reference Hierarchies (Optional)

If your target supports parallel builds for large model reference hierarchies, you can additionally set up automatic verification of MATLAB Distributed Computing Server workers. This addresses the possibility that parallel workers might have different configurations, some of which might not be compatible with a specific target build. For example, the required compiler might not be installed on a worker system.

The code generator provides a programming interface that you can use to automatically check the configuration of parallel workers. If parallel workers are not set up as expected, take action, such as reverting to sequential builds or throwing an error.

To set up automatic verification of workers, you must define a parallel configuration check function named *`STF`*`_par_cfg_chk`, where *`STF`* designates your system target file name. For example, the parallel configuration check function for `ert.tlc` is `ert_par_cfg_chk`.

The general syntax for the function is:

```
function varargout = STF_par_cfg_chk(action,varargin)
```

The number of output and input arguments vary according to the *`action`* specified, and according to the types of information you choose to coordinate between the client and the workers. The function should support the following general sequence of parallel configuration setup calls, differentiated by the first argument passed in:

Call Syntax	Called on:	Action
`cfg = STF_par_cfg_chk ('getPreferredCfg');`	Client	Return a structure representing the preferred configuration for MATLAB Distributed Computing Server workers.
`[tf, cfg] = STF_par_cfg_chk ('getWorkerCfg', cfg);`	Workers	Each worker is passed the MATLAB Distributed Computing Server client's preferred configuration. Return true if the worker can support the preferred configuration; otherwise return false along with a structure representing a configuration the worker can support. Information returned by each worker is added to a cell array of configurations.
`[tf, cfg] = STF_par_cfg_chk ('getCommonCfg', cfgs);`	Client	The client is passed the cell array of worker configurations. If a usable common configuration exists, return true, and return the common configuration to set for all systems. If a common configuration cannot be established, return false or take some action, such as reverting to sequential builds or throwing an error.
`tf = STF_par_cfg_chk ('setCommonCfg', cfg);`	Workers and client	Each system is passed the common configuration to use. Set up the common configuration and, if successful, return true. If errors or issues occur, return false or take some action, such as reverting to sequential builds or throwing an error.
`STF_par_cfg_chk ('clearCfg');`	Workers and client	Clean up after completion of the parallel build.

The parallel configuration check functions for MathWorks provided targets are implemented as wrapper functions that call a function named `parallelMdlRefHostConfigCheckFcn`. For example, see the ERT parallel configuration check function in the file *`matlabroot`*`/toolbox/rtw/rtw/`

`ert_par_cfg_chk.m`, and the function it calls in the file *`matlabroot`*`/toolbox/simulink/simulink/+Simulink/parallelMdlRefHostConfigCheckFcn.m`. The `parallelMdlRefHostConfigCheckFcn` function tries to establish a common compiler across the MATLAB Distributed Computing Server client and workers.

For more information about parallel builds, see "Reduce Build Time for Referenced Models" (Simulink Coder).

Preventing Resource Conflicts (Optional)

Hook files are optional TLC and MATLAB program files that are invoked at well-defined stages of the build process. Hook files let you customize the build process and communicate information between various phases of the process.

If you are adapting your custom target for code generation compatibility with model reference features, consider adding checks to your hook files for handling referenced models differently than top models to prevent resource conflicts.

For example, consider adding the following check to your *`STF`*`_make_rtw_hook.m file`:

```
% Check if this is a referenced model
mdlRefTargetType = get_param(codeGenModelName,`ModelReferenceTargetType');
isNotModelRefTarget = strcmp(mdlRefTargetType, `NONE'); % NONE, SIM, or RTW
if isNotModelRefTarget
  % code that is specific to the top model
else
  % code that is specific to the referenced model
end
```

You may need to do a similar check in your TLC code.

```
%if !IsModelReferenceTarget()
  %% code that is specific to the top model
%else
  %% code that is specific to the referenced model
%endif
```

More About

- "About Embedded Target Development" (Simulink Coder)
- "Sample Custom Targets" (Simulink Coder)
- "Customize System Target Files" (Simulink Coder)

Support Compiler Optimization Level Control

This section describes how to configure a custom embedded target to support compiler optimization level control. Without the described modifications, you will not be able to use the **Compiler optimization level** parameter on the **All Parameters** tab of the Configuration Parameters dialog box to control the compiler optimization level for building generated code. (For more information about compiler optimization level control, see "Compiler optimization level" (Simulink Coder).)

In this section...

About Compiler Optimization Level Control and Custom Targets

The requirements for supporting compiler optimization level control are as follows:

- The target must be derived from the GRT target or the ERT target.
- The system target file (STF) must declare compiler optimization level control compliance, as described in "Declaring Compiler Optimization Level Control Compliance" on page 71-95.
- The target makefile must honor the setting for **Compiler optimization level**, as described in "Providing Compiler Optimization Level Control Support in the Target Makefile" on page 71-96.

For an example that shows how to configure custom target optional features, see "Customize System Target Files" (Simulink Coder).

Declaring Compiler Optimization Level Control Compliance

To declare compiler optimization level control compliance for your target, you must implement a callback function that sets the `CompOptLevelCompliant` flag, and then install the callback function in the `SelectCallback` field of the `rtwgensettings` structure in your STF. The callback function is triggered whenever the user selects the target in the System Target File Browser. For example, the following STF code installs a `SelectCallback` function named `custom_select_callback_handler`:

```
rtwgensettings.SelectCallback = 'custom_select_callback_handler(hDlg,hSrc)';
```

The arguments to the `SelectCallback` function (`hDlg`, `hSrc`) are handles to private data used by the callback API functions. These handles are restricted to use in STF callback functions. They should be passed in without alteration.

Your callback function should set the `CompOptLevelCompliant` flag as follows:

```
slConfigUISetVal(hDlg,hSrc,'CompOptLevelCompliant','on');
slConfigUISetEnabled(hDlg,hSrc,'CompOptLevelCompliant',false);
```

For more information about the STF callback API, see the `slConfigUIGetVal`, `slConfigUISetEnabled`, and `slConfigUISetVal` function reference pages.

When the `CompOptLevelCompliant` target configuration parameter is set to `on`, the **Compiler optimization level** parameter is displayed in the **Code Generation** pane of the Configuration Parameters dialog box for your model.

Providing Compiler Optimization Level Control Support in the Target Makefile

As part of supporting compiler optimization level control for your target, you must modify the target makefile to honor the setting for **Compiler optimization level**. Use a GRT or ERT target provided by MathWorks as a model for making the modifications.

More About

- “About Embedded Target Development” (Simulink Coder)
- “Sample Custom Targets” (Simulink Coder)
- “Customize System Target Files” (Simulink Coder)

Support C Function Prototype Control

This section describes how to configure a custom embedded target to support C function prototype control. Without the described modifications, you will not be able to use the **Configure Model Functions** button on the **Interface** pane of the Configuration Parameters dialog box to control the function prototypes of initialize and step functions that are generated for your model. For more information about C function prototype control, see "Control Generation of Function Prototypes" on page 26-2.

In this section...

About C Function Prototype Control and Custom Targets

The requirements for supporting C function prototype control are as follows:

- The target must be derived from the ERT target.
- The system target file (STF) must declare C function prototype control compliance, as described in "Declaring C Function Prototype Control Compliance" on page 71-97.
- If your target uses a custom static main program, and if a nondefault function prototype control configuration is associated with a model, the static main program must call the function prototype controlled initialize and step functions, as described in "Providing C Function Prototype Control Support in the Custom Static Main Program" on page 71-98.

For an example that shows how to configure custom target optional features, see "Customize System Target Files" (Simulink Coder).

Declaring C Function Prototype Control Compliance

To declare C function prototype control compliance for your target, you must implement a callback function that sets the `ModelStepFunctionPrototypeControlCompliant` flag, and then install the callback function in the `SelectCallback` field of the `rtwgensettings` structure in your STF. The callback function is triggered

whenever the user selects the target in the System Target File Browser. For example, the following STF code installs a `SelectCallback` function named `custom_select_callback_handler`:

```
rtwgensettings.SelectCallback = 'custom_select_callback_handler(hDlg,hSrc)';
```

The arguments to the `SelectCallback` function (`hDlg`, `hSrc`) are handles to private data used by the callback API functions. These handles are restricted to use in STF callback functions. They should be passed in without alteration.

Your callback function should set the `ModelStepFunctionPrototypeControlCompliant` flag as follows:

```
slConfigUISetVal(hDlg,hSrc,'ModelStepFunctionPrototypeControlCompliant','on');
slConfigUISetEnabled(hDlg,hSrc,'ModelStepFunctionPrototypeControlCompliant',false);
```

For more information about the STF callback API, see the `slConfigUIGetVal`, `slConfigUISetEnabled`, and `slConfigUISetVal` function reference pages.

When the `ModelStepFunctionPrototypeControlCompliant` target configuration parameter is set to `on`, you can use the **Configure Model Functions** button on the **Interface** pane of the Configuration Parameters dialog box to control the function prototypes of initialize and step functions that are generated for your model.

Providing C Function Prototype Control Support in the Custom Static Main Program

If your target uses a custom static main program, and if a nondefault function prototype control configuration is associated with a model, you must update the static main program to call the function prototype controlled initialize and step functions. You can do this in either of the following ways:

1. Manually adapt your static main program to declare model data and call the function prototype controlled initialize and step functions.
2. Generate your main program using **Generate an example main program** on the **Templates** pane of the Configuration Parameters dialog box. The generated main program declares model data and calls the function prototype controlled initialize and step function.

More About

- "About Embedded Target Development" (Simulink Coder)

- “Sample Custom Targets” (Simulink Coder)
- “Customize System Target Files” (Simulink Coder)

Support C++ Class Interface Control

This section describes how to configure a custom embedded target to support C++ class interface control. Without the described modifications, you will not be able to use `C++ class` code interface packaging and the **Configure C++ Class Interface** button on the **Interface** pane of the Configuration Parameters dialog box to generate and configure C++ class interfaces to model code. For more information about C++ class interface control, see "Control Generation of C++ Class Interfaces" on page 26-23.

In this section...

About C++ Class Interface Control and Custom Targets

The requirements for supporting C++ class interface control are as follows:

- The target must be derived from the ERT target.
- The system target file (STF) must declare C++ class interface control compliance, as described in "Declaring C++ Class Interface Control Compliance" on page 71-100.

For an example that shows how to configure custom target optional features, see "Customize System Target Files" (Simulink Coder).

Declaring C++ Class Interface Control Compliance

To declare C++ class interface control compliance for your target, you must implement a callback function that sets the `CPPClassGenCompliant` flag, and then install the callback function in the `SelectCallback` field of the `rtwgensettings` structure in your STF. The callback function is triggered whenever the user selects the target in the System Target File Browser. For example, the following STF code installs a `SelectCallback` function named `custom_select_callback_handler`:

```
rtwgensettings.SelectCallback = 'custom_select_callback_handler(hDlg,hSrc)';
```

The arguments to the `SelectCallback` function (`hDlg`, `hSrc`) are handles to private data used by the callback API functions. These handles are restricted to use in STF callback functions. They should be passed in without alteration.

Your callback function should set the `CPPClassGenCompliant` flag as follows:

```
slConfigUISetVal(hDlg,hSrc,'CPPClassGenCompliant','on');
slConfigUISetEnabled(hDlg,hSrc,'CPPClassGenCompliant',false);
```

For more information about the STF callback API, see the `slConfigUIGetVal`, `slConfigUISetEnabled`, and `slConfigUISetVal` function reference pages.

When the `CPPClassGenCompliant` target configuration parameter is set to `on`, you can use the `C++ class` code interface packaging and the **Configure C++ Class Interface** button on the **Interface** pane of the Configuration Parameters dialog box to generate and configure C++ class interfaces to model code.

Providing C++ Class Interface Control Support in the Custom Static Main Program

Selecting `C++ class` code interface packaging for your model turns on the model option **Generate an example main program**. With this option on, code generation generates an example main program, `ert_main.cpp`. The generated example main program declares model data and calls the C++ class interface configured model step method, and illustrates how the generated code can be deployed.

To customize the build process and disable generation and inclusion of an example main program, see the `setTargetProvidesMain` function. Disabling example main generation permits including a custom main program.

More About

- “About Embedded Target Development” (Simulink Coder)
- “Sample Custom Targets” (Simulink Coder)
- “Customize System Target Files” (Simulink Coder)

Support Concurrent Execution of Multiple Tasks

If a custom embedded target must support concurrent execution of multiple tasks on a multicore platform, the target must declare support for concurrent execution by setting the target configuration option `ConcurrentExecutionCompliant`. Otherwise, you will not be able to configure a multicore target model for concurrent execution.

If `ConcurrentExecutionCompliant` is not already configured for your custom target, you can set the option in the following ways:

- Include the following code directly in your system target file (*`mytarget`*`.tlc`):

  ```
  rtwgensettings.SelectCallback = 'slConfigUISetVal(hDlg,hSrc,...
  ''ConcurrentExecutionCompliant'',''on'');';
  rtwgensettings.ActivateCallback = 'slConfigUISetVal(hDlg,hSrc,...
  ''ConcurrentExecutionCompliant'',''on'');';
  ```

- Implement a callback function that sets the `ConcurrentExecutionCompliant` option, and then install the callback function in the `SelectCallback` field of the `rtwgensettings` structure in your STF. The callback function is triggered whenever the user selects the target in the System Target File Browser. For example, the following STF code installs a `SelectCallback` function named `custom_select_callback_handler`:

  ```
  rtwgensettings.SelectCallback = 'custom_select_callback_handler(hDlg,hSrc)';
  ```

  The arguments to the `SelectCallback` function (`hDlg`, `hSrc`) are handles to private data used by the callback API functions. These handles are restricted to use in STF callback functions. They should be passed in without alteration.

  Your callback function should set the `ConcurrentExecutionCompliant` option as follows:

  ```
  slConfigUISetVal(hDlg,hSrc,'ConcurrentExecutionCompliant','on');
  slConfigUISetEnabled(hDlg,hSrc,'ConcurrentExecutionCompliant',false);
  ```

  For more information about the STF callback API, see the `slConfigUIGetVal`, `slConfigUISetEnabled`, and `slConfigUISetVal` function reference pages.

When the `ConcurrentExecutionCompliant` target configuration option is set to `'on'`, you can select the custom target and configure your multicore target model for concurrent execution.

For an example that shows how to configure custom target optional features, see "Customize System Target Files" (Simulink Coder).

More About

- “About Embedded Target Development” (Simulink Coder)
- “Sample Custom Targets” (Simulink Coder)
- “Customize System Target Files” (Simulink Coder)

Interface to Development Tools

Unless you are developing a target purely for code generation purposes, you will want your embedded target to support a complete build process. A full post-code generation build process includes

- Compilation of generated code
- Linking of compiled code and runtime libraries into an executable program module (or some intermediate representation of the executable code, such as S-Rec format)
- Downloading the executable to target hardware with a debugger or other utility
- Initiating execution of the downloaded program

Supporting a complete build process is inherently a complex task, because it involves interfacing to cross-development tools and utilities that are external to the code generator.

If your development tools can be controlled with traditional makefiles and a make utility such as `gmake`, it may be relatively simple for you to adapt existing target files (such as the `ert.tlc` and `ert.tmf` files provided by the Embedded Coder software) to your requirements. This approach is discussed in "Template Makefile Approach" on page 71-105.

In this section...

About Interfacing to Development Tools

Automating your build process through a modern integrated development environment (IDE) presents a different set of challenges. Each IDE has its own way of representing the set of source files and libraries for a project and for specifying build arguments. Interfacing to an IDE may require generation of specialized file formats required by the IDE (for example, project files) and, and also may require the use of inter-application communication (IAC) techniques to run the IDE. One such approach to build automation is discussed in "Interface to an Integrated Development Environment" on page 71-105.

Template Makefile Approach

A template makefile provides information about your model and your development system. The build process uses this information to create a makefile (`.mk` file) to build an executable program. The code generator provides a number of template makefiles suitable for development computer compilers, such as LCC (`ert_lcc.tmf`) and Microsoft Visual C++ (`ert_vc.tmf`).

Adapting one of the existing template makefiles to your cross-compiler's make utility may require little more than copying and renaming the template makefile in accordance with the conventions of your project.

If you need to make more extensive modifications, you need to understand template makefiles in detail. For a detailed description of the structure of template makefiles and of the tokens used in template makefiles, see "Customize Template Makefiles" on page 71-62.

The following topics supplement the basic template makefile information:

- "Supporting Multiple Development Environments" on page 71-48
- "Supplying Development Environment Information to Your Template Makefile" on page 71-26

Interface to an Integrated Development Environment

This section describes techniques that have been used to integrate embedded targets with integrated development environment (IDEs), including

- How to generate a header file containing directives to define variables (and their values) required by a non-makefile based build.
- Some problems and solutions specific to interfacing embedded targets with the Freescale Semiconductor CodeWarrior IDE. The examples provided should help you to deal with similar interfacing problems with your particular IDE.

- "Generating a CPP_REQ_DEFINES Header File" on page 71-105
- "Interfacing to the Freescale CodeWarrior IDE" on page 71-107

Generating a CPP_REQ_DEFINES Header File

In template makefiles, the token `CPP_REQ_DEFINES` is expanded and replaced with a list of parameter settings entered with various dialog boxes. This variable often contains

information such as MODEL (name of generating model), NUMST (number of sample times in the model), MT (model is multitasking or not), and numerous other parameters (see "Template Makefiles and Tokens" on page 71-62).

The makefile mechanism handles the CPP_REQ_DEFINES token automatically. If your target requires use of a project file, rather than the traditional makefile approach, you can generate a header file containing directives to define these variables and provide their values.

The following TLC file, gen_rtw_req_defines.tlc, provides an example. The code generates a C header file, cpp_req_defines.h. The information required to generate each #define directive is derived either from information in the *model*.rtw file (e.g., CompiledModel.NumSynchronousSampleTimes), or from make variables from the rtwoptions structure (for example, PurelyIntegerCode).

```
%% File: gen_rtw_req_defines_h.tlc
%openfile CPP_DEFINES = "cpp_req_defines.h"
#ifndef _CPP_REQ_DEFINES_
#define _CPP_REQ_DEFINES_
#define MODEL %<CompiledModel.Name>
#define ERT  1
#define NUMST %<CompiledModel.NumSynchronousSampleTimes>
#define TID01EQ %<CompiledModel.FixedStepOpts.TID01EQ>
%%
%if CompiledModel.FixedStepOpts.SolverMode == "MultiTasking"
#define MT  1
#define MULTITASKING  1
%else
#define MT  0
#define MULTITASKING  0
%endif
%%
#define MAT_FILE  0
#define INTEGER_CODE %<PurelyIntegerCode>
#define ONESTEPFCN %<CombineOutputUpdateFcns>
#define TERMFCN %<IncludeMdlTerminateFcn>
%%
#define MULTI_INSTANCE_CODE  0
#define HAVESTDIO  0
#endif
%closefile CPP_DEFINES
```

Interfacing to the Freescale CodeWarrior IDE

Interfacing an embedded target's build process to the CodeWarrior IDE requires that two problems must be dealt with:

- The build process must generate a CodeWarrior compatible project file. This problem, and a solution, is discussed in "XML Project Import" on page 71-107. The solution described is applicable to ASCII project file formats.
- During code generation, the target must automate a CodeWarrior session that opens a project file and builds an executable. This task is described in "Build Process Automation" on page 71-112. The solution described is applicable to IDEs that can be controlled with Microsoft Component Object Model (COM) automation.

XML Project Import

This section illustrates how to use the Target Language Compiler (TLC) to generate an eXtensible Markup Language (XML) file, suitable for import into the CodeWarrior IDE, that contains information about the source code generated by an embedded target.

The choice of XML format is dictated by the fact that the CodeWarrior IDE supports project export and import with XML files. As of this writing, native CodeWarrior project files are in a proprietary binary format.

Note that if your target needs to support some other compiler's project file format, you can apply the techniques shown here to other ASCII file formats (see "Generating a CPP_REQ_DEFINES Header File" on page 71-105).

To illustrate the basic concept, consider a hypothetical XML file exported from a CodeWarrior stationery project. The following is a partial listing:

```
<target>
   <settings>
    ...
   <\settings>
   <file><name>foo.c<\name>
   <\file>
   ...
   <file><name>foobar.c<\name>
   <\file>
   <fileref><name>foo.c<\name>
   <\fileref>
   ...
   <fileref><name>foobar.c<\name>
```

```
   <\fileref>
<\target>
```

Insert this XML code into an `%openfile/%closefile` block within a TLC file, `test.tlc`, as shown below.

```
%% test.tlc
%% This code will generate a file model_project.xml,
%% where model is the generating model name specified in
%% the CompiledModel.Name field of the model.rtw file.
%openfile XMLFileContents = %<CompiledModel.Name>_project.xml
<target>
   <settings>
    ...
   <\settings>
   <file><name>%<CompiledModel.Name>.c<\name>
   <\file>
   ...
   <file><name>foobar.c<\name>
   <\file>
   <fileref><name>%<CompiledModel.Name>.c<\name>
   <\fileref>
   ...
   <fileref><name>foobar.c<\name>
   <\fileref>
<\target>
%closefile XMLFileContents
%selectfile NULL_FILE
```

Note the use of the TLC token `CompiledModel.Name`. The token is resolved and the resulting filename is included in the output stream. You can specify other information, such as paths and libraries, in the output stream by specifying other tokens defined in *`model`*`.rtw`. For example, `System.Name` may be defined as `<Root>/Subsystem1`.

Now suppose that `test.tlc` is invoked during a target's build process, where the generating model is `mymodel`. This should be done after the `codegenentry` statement. For example, `test.tlc` could be included directly in the system target file:

```
%include "codegenentry.tlc"
%include "test.tlc"
```

Alternatively, the `%include "test.tlc"` directive could be inserted into the `mytarget_genfiles.tlc` hook file, if present.

TLC tokens such as

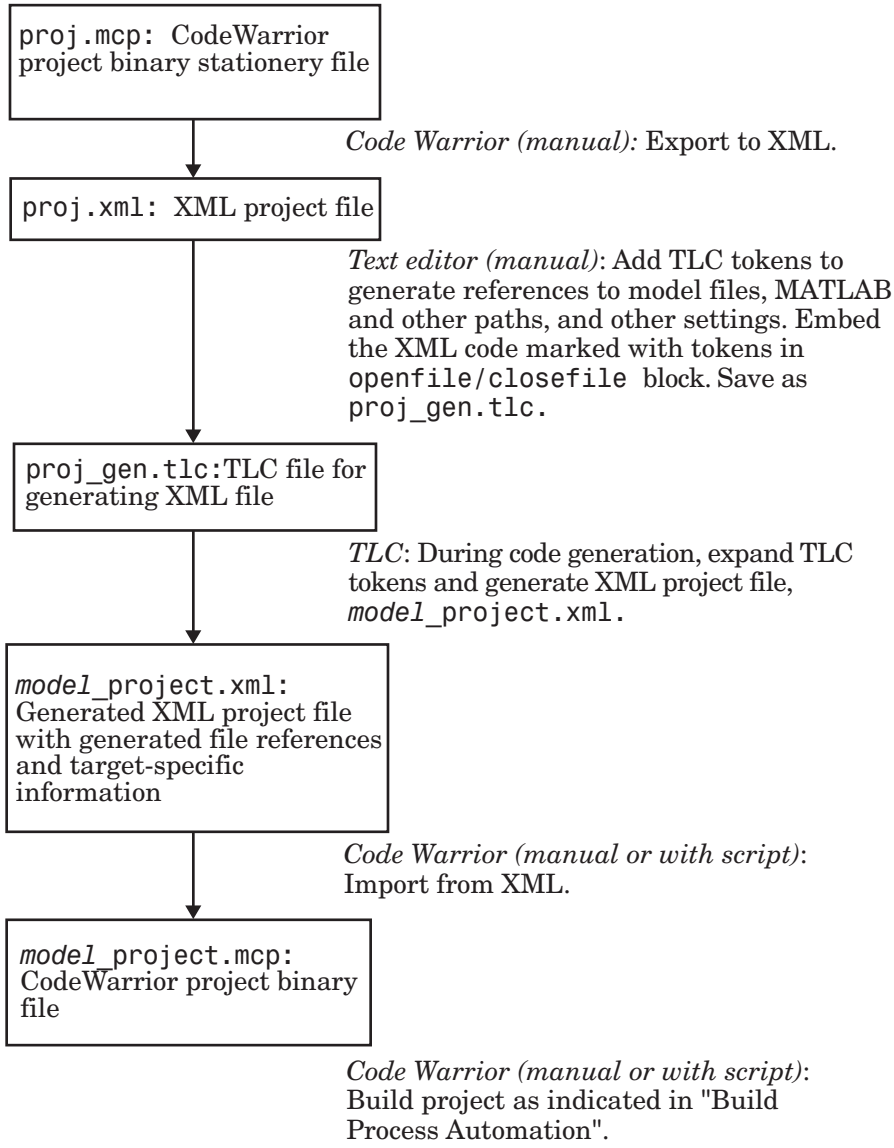
```
<file><name>%<CompiledModel.Name>.c<\name>
```

are expanded, with the `CompiledModel` record in the `mymodel.rtw` file, as in

```
<file><name>mymodel.c<\name>
```

`test.tlc` generates an XML file, file *`model`*`_project.xml`, from a model. *`model`*`_project.xml` contains references to generated code files. *`model`*`_project.xml` can be imported into the CodeWarrior IDE as a project.

The following flowchart summarizes this process.



Note This process has drawbacks. First, manually editing an XML file exported from a CodeWarrior stationery project can be a laborious task, involving modification of a few

dozen lines embedded within several thousand lines of XML code. Second, if you make changes to the CodeWarrior project after importing the generated XML file, the XML file must be exported and manually edited once again.

Build Process Automation

An application that supports COM automation can control other applications that include a COM interface. Using MATLAB COM automation functions, a MATLAB file can command a COM-compatible development system to execute tasks required by the build process.

The MATLAB COM automation functions described in this section are documented in "Call COM Objects" (MATLAB).

For information about automation commands supported by the CodeWarrior IDE, see your CodeWarrior documentation.

COM automation is used by some embedded targets to automate the CodeWarrior IDE to execute tasks such as:

- Opening a new CodeWarrior session
- Configure a project
- Loading a CodeWarrior project file
- Removing object code from the project
- Building or rebuilding the project
- Debug an application

COM technology automates certain repetitive tasks and allows the user to interact directly with the external application. For example, when the end user of the embedded targets capability initiates a build, the target quickly invokes CodeWarrior actions and leaves a project built and ready to run with the IDE.

Example COM Automation Functions

The functions below use the MATLAB `actxserver` command to invoke COM functions for controlling the CodeWarrior IDE from a MATLAB file:

- `CreateCWComObject`: Create a COM connection to the CodeWarrior IDE.
- `OpenCW`: Open the CodeWarrior IDE without opening a project.

- `OpenMCP`: Open the CodeWarrior project file (`.mcp` file) specified by the input argument.
- `BuildCW`: Open the specified `.mcp` file, remove object code, and build project.

These functions are examples; they do not constitute a full implementation of a COM automation interface. If your target creates the project file during code generation, the top-level `BuildCW` function should be called after the code generation process is completed. Normally `BuildCW` would be called from the `exit` method of your *`STF`*`_make_rtw_hook.m` file (see "STF_make_rtw_hook.m" on page 71-21).

In the code examples, the variable `in_qualifiedMCP` is assumed to store a fully qualified path to a CodeWarrior project file (for example, path, filename, and extension). For example:

```
in_qualifiedMCP = 'd:\work\myproject.mcp';
```

In actual practice, your code is responsible for determining the conventions used for the project filename and location. One simple convention would be to default to a project file *`model`*`.mcp`, located in your target's build folder.

```
%=============================================================================
% Function: CreateCWComObject
% Abstract: Creates the COM connection to CodeWarrior
%
function ICodeWarriorApp = CreateCWComObject
  vprint([mfilename ': creating CW com object']);
  try
    ICodeWarriorApp = actxserver('CodeWarrior.CodeWarriorApp');
  catch
    error(['Error creating COM connection to ' ComObj ...
       '. Verify that CodeWarrior is installed. Verify COM access to
CodeWarrior outside of MATLAB.']);
  end
  return;


%=============================================================================
% Function: OpenCW
% Abstract: Opens CodeWarrior without opening a project. Returns the
%           handle ICodeWarriorApp.
%
function ICodeWarriorApp = OpenCW()
  ICodeWarriorApp = CreateCWComObject;
  CloseAll;
  OpenMCP(in_qualifiedMCP);


%=============================================================================
% Function: OpenMCP
```

```
% Abstract: open an MCP project file
%
function OpenMCP(in_qualifiedMCP)
  % Argument checking. This method requires valid project file.
  if ~exist(in_qualifiedMCP)
      error([mfilename ': Missing or empty project file argument']);
  end
  if isempty(in_qualifiedMCP)
      error([mfilename ': Missing or empty project file argument']);
  end
  ICodeWarriorApp = CreateCWComObject;
  vprint([mfilename ': Importing']);
  try
    ICodeWarriorProject = ...
      invoke(ICodeWarriorApp.Application,...
      'OpenProject', in_qualifiedMCP,...
      1,0,0);
  catch
    error(['Error using COM connection to import project. ' ...
       ' Verify that CodeWarrior is installed. Verify COM access to
CodeWarrior outside of MATLAB.']);
  end


%=========================================================================
% Function: BuildCW
% Abstract: Opens CodeWarrior.
%           Opens the specified CodeWarrior project.
%           Deletes objects.
%           Builds.
%
function ICodeWarriorApp = BuildCW(in_qualifiedMCP)
    % ICodeWarriorApp = BuildCW;
    ICodeWarriorApp = CreateCWComObject;
    CloseAll;
    OpenMCP(in_qualifiedMCP);
    try
      invoke(ICodeWarriorApp.DefaultProject,'RemoveObjectCode', 0, 1);
    catch
      error(['Error using COM connection to remove objects of current project. ' ...
       'Verify that CodeWarrior is installed. Verify COM access to
CodeWarrior outside of MATLAB.']);
    end
    try
      invoke(ICodeWarriorApp.DefaultProject,'BuildAndWaitToComplete');
    catch
      error(['Error using COM connection to build current project. ' ...
       'Verify that CodeWarrior is installed. Verify COM access to
CodeWarrior outside of MATLAB.']);
    end
```

More About

- “About Embedded Target Development” (Simulink Coder)

- “Sample Custom Targets” (Simulink Coder)
- “Customize System Target Files” (Simulink Coder)

Device Drivers

Device drivers that communicate with target hardware are essential to many real-time development projects.

You can integrate existing C (or C++) device driver functions into Simulink models by using the Legacy Code Tool. When you use the code generator to generate code from a model, the Legacy Code Tool can insert a call to your C function into the generated code. For details, see “Import Calls to External Code into Generated Code with Legacy Code Tool” (Simulink Coder) and “Integrate C Functions into Simulink Models with Legacy Code Tool” (Simulink).

More About

- “About Embedded Target Development” (Simulink Coder)
- “Sample Custom Targets” (Simulink Coder)
- “Customize System Target Files” (Simulink Coder)

Project and Build Configurations for Embedded Targets in Embedded Coder

Model Setup

In this section...

Block Selection

You can create models for targeting the same way you create other Simulink models—by combining standard blocks and C-MEX S-functions.

You can use blocks from the following sources:

- The Embedded Coder Support Packages.
- The Embedded Targets library (`embeddedtargetslib`) in the Embedded Coder product.
- Blocks from the System Toolboxes products
- Custom blocks

Avoid using blocks that do not generate code, including the following blocks.

Block Name/Category	Library	Description
Scope	Simulink, DSP System Toolbox software	Provides oscilloscope view of your output. Do not use the **Save data to workspace** option on the **Data history** pane in the Scope Parameters dialog.
To Workspace	Simulink	Return data to your MATLAB workspace.
From Workspace	Simulink	Send data to your model from your MATLAB workspace.
Spectrum Scope	DSP System Toolbox	Compute and display the short-time FFT of a signal. It has internal buffering that can slow your process without adding value.

Block Name/Category	Library	Description
To File	Simulink	Send data to a file on your host machine.
From File	Simulink	Get data from a file on your host machine.
Triggered to Workspace	DSP System Toolbox	Send data to your MATLAB workspace.
Signal To Workspace	DSP System Toolbox	Send a signal to your MATLAB workspace.
Signal From Workspace	DSP System Toolbox	Get a signal from your MATLAB workspace.
Triggered Signal From Workspace	DSP System Toolbox	Get a signal from your MATLAB workspace.
To Wave device	DSP System Toolbox	Send data to a `.wav` device.
From Wave device	DSP System Toolbox	Get data from a `.wav` device.

Configure Target Hardware Resources

This topic contains the following subtopics:

About Supported IDEs

This "Configure Target Hardware Resources" on page 72-3 section applies to the following IDEs:

- Analog Devices VisualDSP++®
- Wind River Diab/GCC (makefile generation only)

Configure Parameters Under the Target Hardware Resources Tab

Configure the parameters under the Target Hardware Resources tab of your Simulink model for a specific tool chain and target hardware. Doing so updates other parameters in the Configuration Parameters dialog to the default values for the software build tool chain and target hardware you are using.

Note: The Target Preferences (Removed) block has been removed from the Simulink block libraries for the Embedded Coder and Simulink Coder products.

Parameters in the Target Preferences block have been moved to the Target Hardware Resources tab.

To configure your Simulink model for a specific tool chain and target hardware:

1. In a Simulink model, open the model Configuration Parameters by:
   - Clicking the gear icon,

   - Pressing **Ctrl+E** on your keyboard
   - Selecting the **Simulation** > **Model Configuration Parameters** menu items
2. In the Configuration Parameters dialog, click **Code Generation**, and then click "+" next to Code Generation. This action displays the sub-panes under Code Generation.
3. On the Code Generation pane, change **System target file** to `idelink_ert.tlc` or `idelink_grt.tlc`.

   The dialog displays a **Coder Target** pane under the Code Generation pane.
4. Select the **Coder Target** pane.
5. Select the **Target Hardware Resources** tab.
6. Set the following parameters to match the tool chain and target hardware you are using:
   - **IDE/Tool Chain**
   - **Board**
   - **Processor**
7. Review the other parameters under the **Target Hardware Resources** tab.
8. Click **Apply**, and save the changes to your model.

Configuration Parameters

- "What are Configuration Parameters?" on page 72-5

What are Configuration Parameters?

To see the model Configuration Parameters, open the **Configuration Parameters** dialog. You can do this in the model editor by selecting **Simulation** > **Model Configuration Parameters**, or by pressing **Ctrl+E** on your keyboard.

The **Configuration Parameters** dialog specifies the values for a model's active *configuration set*. These parameters determine the type of solver used, the import and export settings, and other values that determine how the model runs.

Setting Model Configuration Parameters

To set the Configuration Parameters to the right values for you to generate code from your model, see "Configure Parameters Under the Target Hardware Resources Tab" on page 72-3. This action initializes the model Configuration Parameters to the right default values for you to generate code. You can then use the Configuration Parameters dialog to make further modifications to the values. You can generate buildable code using these default values.

The following subsections provide a quick overview of the panes and parameters with which you are most likely to interact.

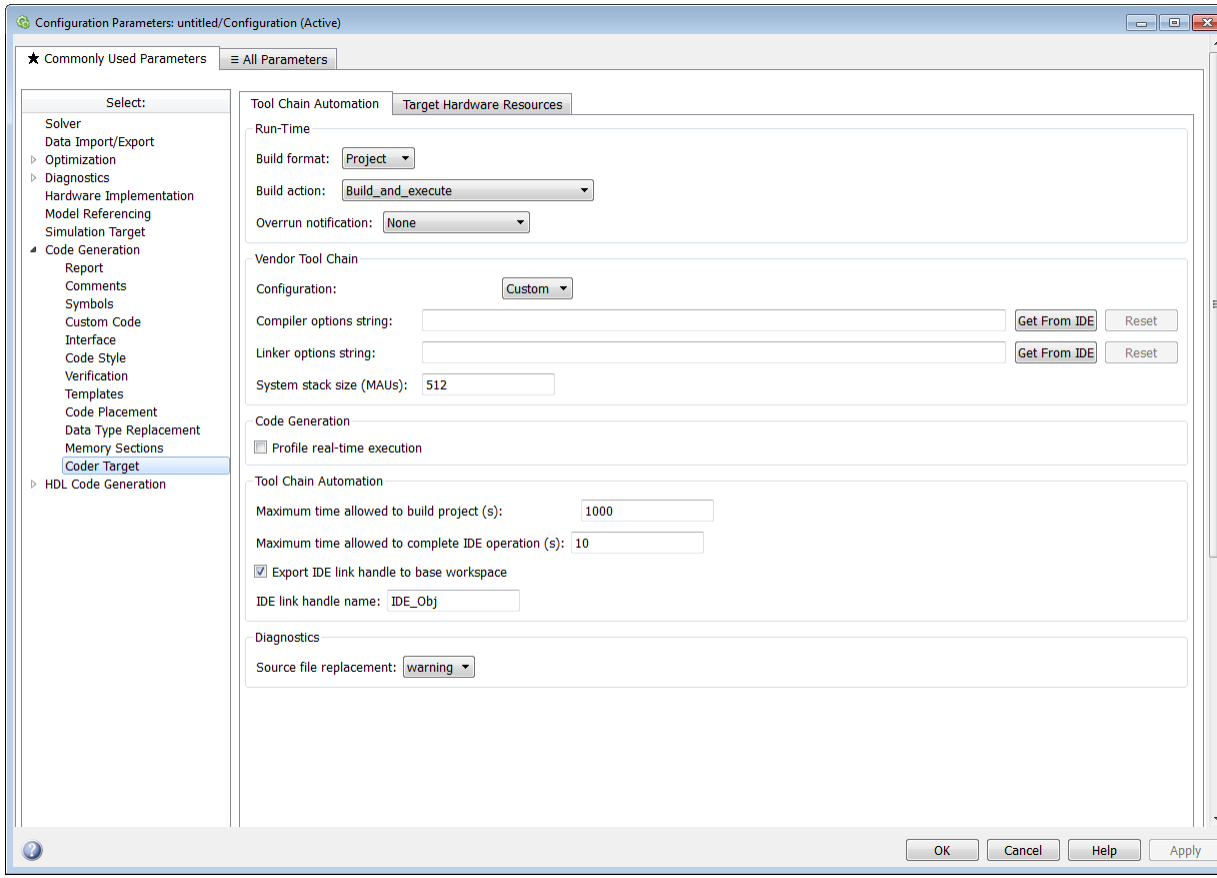
Code Generation Pane

When you set **System target file** to `idelink_ert.tlc` or `idelink_grt.tlc`, the dialog adds an **Coder Target** pane to the list of panes under **Code Generation**.

Leave **Language** set to `C`. The `idelink_ert.tlc` and `idelink_grt.tlc` system target files do not support `C++` code generation.

For more information, see "Model Configuration Parameters: Code Generation" (Simulink Coder).

Coder Target Pane Parameters



The Coder Target entry provides options in these areas:

- **Run-Time** — Set options for run-time operations, like the build action
- **Vendor Tool Chain** — Set compiler, linker, and system stack size options
- **Code Generation** — Configure your code generation requirements
- **Link Automation** — Export an IDE link handle object, such as `IDE_Obj`, to your MATLAB workspace
- **Diagnostics** — Determine how the code generation process responds when you use source code replacement in the **Custom Code** pane.

For more information, see Code Generation Pane: Coder Target.

Build format

Select `Project` to create an IDE project, or select `Makefile` to create a makefile build script.

Build action

Your selection for **Build action** determines what happens when you press **Ctrl+B**. Your selection tells Simulink Coder software when to stop the code generation and build process.

To run your model on the processor, select `Build_and_execute`. This selection is the default build action.

The actions are cumulative—each action performs an additional step relative to the preceding action on the list.

If you set **Build format** to `Project`, select one of the following options:

- `Create_project` — Directs Simulink Coder software to start the IDE and populate a new project with the files from the build process. This option offers a convenient way to build projects in the IDE.
- `Archive_library` — Directs Simulink Coder software to create an archive library for this model. Use this option when you plan to use the model in a model reference application. Model reference requires that you archive your the IDE projects for models that you use in model referencing.
- `Build` — Builds the executable file, but does not download the file to the target hardware.
- `Build_and_execute` — Directs Simulink Coder software to build, download, and run your generated code as an executable on your target hardware.
- `Create_processor_in_the_loop_project` — Directs code generation process to create PIL algorithm object code as part of the project build. This option requires an Embedded Coder license.

If you set **Build format** to `Makefile`, select one of the following options:

- `Create_makefile` — Creates a makefile.
- `Archive_library` — Creates a makefile and the generated output will be an archive library.

- `Build` — Creates a makefile and an executable.
- `Build_and_execute` — Creates a makefile and an executable. Then it evaluates the execute instruction in the current configuration.

Overrun notification

To enable the overrun indicator, choose one of three ways for the target to respond to an overrun condition in your model:

- `None` — Ignore overruns encountered while running the model.
- `Print_message` — When the target encounters an overrun condition, it prints a message to the standard output device, `stdout`.
- `Call_custom_function` — Respond to overrun conditions by calling the custom function you identify in **Function name**.

Function name

When you select `Call_custom_function` from the **Overrun notification** list, you enable this option. Enter the name of the function the target should use to notify you that an overrun condition occurred. The function must exist in your code on the target hardware.

Configuration

The **Configuration** parameter defines sets of build options that apply to the files generated from your model.

The `Release` and `Debug` option apply build settings that are defined by your compiler. For more information, refer to your compiler documentation.

`Custom` has the same default values as `Release`, but:

- Leaves **Compiler options string** empty.

Compiler options string

To determine the degree of optimization provided by the optimizing compiler, enter the optimization level to apply to files in your project. For details about the compiler options, refer to your IDE documentation. When you create new projects, the code generator does not set optimization flags.

With Texas Instruments Code Composer Studio™ 3.3 and Analog Devices VisualDSP++, the user interface displays **Get From IDE** and **Reset** buttons next to this parameter.

If you have an active project open in the IDE, you can click **Get From IDE** to import the compiler option setting from the current project in the IDE. To reset the compiler option to the default value, click **Reset**.

Linker options string

To specify the options provided by the linker during link time, you enter the linker options as text. For details about the linker options, refer to your IDE documentation. When you create new projects, the code generator does not set linker options.

With Texas Instruments Code Composer Studio 3.3 and Analog Devices VisualDSP++, the user interface displays **Get From IDE** and **Reset** buttons next to this parameter. If you have an active project open in the IDE, you can click **Get From IDE** to import the linker options text from the current project in the IDE. To clear the linker options, click **Reset**.

System stack size (MAUs)

Enter the amount of memory that is available for allocating stack data, measured in minimum addressable units (MAU). Block output buffers are placed on the stack until the stack memory is fully allocated. After that, the output buffers go in global memory. An MAU is typically 1 byte, but its size can vary by target hardware.

This parameter is used in targets to allocate the stack size for the generated application. For example, with embedded processors that are not running an operating system, this parameter determines the total stack space that can be used for the application. For operating systems, this value specifies the stack space allocated per thread.

This parameter also applies to the “Maximum stack size (bytes)” (Simulink) parameter, located in the Optimization > Signals and Parameters pane.

System heap size (MAUs)

Set the default heap size that the target hardware reserves for dynamic memory allocation.

The target hardware uses this heap for functions like printf() and system services code.

The following IDEs use this parameter:

- Analog Devices VisualDSP++
- Wind River Diab/GCC (makefile generation only)

Profile real-time execution

To enable the real-time execution profile capability, select **Profile real-time execution**. With this selected, the build process instruments your code to provide performance profiling at the task level or for atomic subsystems. When you run your code, the executed code reports the profiling information in an HTML report.

Link Automation

When you build a model for a system target file , the code generator automatically creates or uses an existing *IDE link handle object* (named `IDE_Obj`, by default) to connect to your IDE.

Although `IDE_Obj` is a handle for a specific instance of the IDE, it also contains information about the IDE instance to which it refers, such as the target the IDE accesses. In this pane, the **Export IDE link handle to base workspace** option lets you instruct the code generator to export the object to your MATLAB workspace, giving it the name you assign in **IDE link handle name**.

You can also use the IDE link handle object to interact with the IDE using IDE Automation Interface commands.

Maximum time allowed to build project (s)

Specifies how long the software waits for the IDE to build the software.

Maximum time allowed to complete IDE operation (s)

Specifies how long the software waits for IDE functions, such as `read` or `write`, to return completion messages. If you do not specify a timeout, the default value is 10 seconds.

Export IDE link handle to base workspace

Directs the software to export the IDE_Obj object to your MATLAB workspace.

IDE link handle name

Specifies the name of the IDE_Obj object that the build process creates.

Source file replacement

Selects the diagnostic action to take if the software detects conflicts when you replace source code with custom code. The diagnostic message responds to both source file

replacement in the Configuration Parameters under Code Generation > Coder Target parameters and under Code Generation > Custom Code.

The following settings define the messages you see and how the code generation process responds:

- `none` — Does not generate warnings or errors when it finds conflicts.
- `warning` — Displays a warning. `warn` is the default value.
- `error` — Terminates the build process and displays an error message that identifies which file has the problem and suggests how to resolve it.

The build operation continues if you select `warning` and the software detects custom code replacement problems. You see warning messages as the build progresses.

Select `error` the first time you build your project after you specify custom code to use. The error messages can help you diagnose problems with your custom code replacement files. Use `none` when the replacement process works and you do not want to see multiple messages during your build.

Model Reference

The `ert.tlc` system target files provide support for generating code from models that use Model Reference. A referenced model will generate an archive library.

To enable Model Reference builds:

1. Open your referenced model.
2. Select Simulation > Model Configuration Parameters from the model menus.
3. From the list of panes under **Code Generation**, choose **Coder Target**.
4. In the right pane, under Run-Time, select `Archive_library` from the **Build action** list.

If your top-model uses a reference model that does not have the **Build action** set to `Archive_library`, the build process automatically changes the **Build action** to `Archive_library` and issues a warning about the change.

Configuration Parameters in Reference Models

Use the same Coder Target pane settings in Configuration Parameters for the models in the model hierarchy.

XMakefiles for Software Build Tool Chains

In this section...

What is the XMakefile Feature

Overview

You can use makefiles instead of IDE projects during the automated software build process. This approach is described in "Using Makefiles to Generate and Build Software" on page 72-14.

The XMakefile feature lets you choose the *configuration* of a specific software build tool chain to use during the automated build process. The configuration contains paths and settings for your make utility, compiler, linker, archiver, pre-build, post-build, and execute tools.

You can also create a new configuration for a new tool chain, as described in "Creating a New XMakefile Configuration" on page 72-16.

Your requirements for specific features may determine whether you choose makefiles or IDE projects. See "Feature Support" on page 72-13.

Available XMakefile Configurations

The following list describes the configurations in the XMakefile dialog that this product supports:

- `adivdsp_blackfin`: Analog Devices VisualDSP++ & Analog Devices Blackfin®
- `adivdsp_sharc`: Analog Devices VisualDSP++ & Analog Devices SHARC®

- `adivdsp_tigersharc`: Analog Devices VisualDSP++ & Analog Devices TigerSHARC®
- `gcc_target`: GNU Compiler Collection & Host Operating System or Embedded Operating System
- `wrsdiab_arm9_vxworks67_rtp`: Wind River Systems DIAB Compiler & ARM 9 & VxWorks 6.7 & real-time process applications
- `wrsdiab_arm9_vxworks67_rtp_so`: Wind River Systems DIAB Compiler & ARM 9 & VxWorks 6.7 & real-time process applications with shared object
- `wrsdiab_hostsim_vxworks67_rtp`: Wind River Systems DIAB Compiler & VxWorks Host Simulator & VxWorks 6.7 & real-time process applications
- `wrsdiab_hostsim_vxworks67_rtp_so`: Wind River Systems DIAB Compiler & VxWorks Host Simulator & VxWorks 6.7 & real-time process applications with shared object
- `wrsdiab_hostsim_vxworks68_rtp`: Wind River Systems DIAB Compiler & VxWorks Host Simulator & VxWorks 6.8 & real-time process applications
- `wrsdiab_hostsim_vxworks68_rtp_so`: Wind River Systems DIAB Compiler & VxWorks Host Simulator & VxWorks 6.8 & real-time process applications with shared object
- `wrsgnu_arm9_vxworks67_rtp`: Wind River Systems GNU Compiler & VxWorks Host Simulator & VxWorks 6.7 & real-time process applications
- `wrsgnu_hostsim_vxworks67_rtp`: Wind River Systems GNU Compiler & VxWorks Host Simulator & VxWorks 6.7 & real-time process applications with shared object
- `wrsgnu_hostsim_vxworks68_rtp`: Wind River Systems GNU Compiler & VxWorks Host Simulator & VxWorks 6.8 & real-time process applications with shared object
- `xilinx_ise_14_x`: Xilinx ISE Design Suite & ARM Cortex-A9 running Linux on Xilinx Zynq®-7000 platform

For more information about supported versions of third-party software, see

Feature Support

With makefiles, you cannot use features that rely on direct communications between your MathWorks software and third-party IDEs.

You cannot use the following features with makefiles:

- IDE Project Generation

- IDE Automation Interface
- IDE debugger communications during Processor-in-the-loop (PIL) simulation

Using Makefiles to Generate and Build Software

Configuring Your Model to Use Makefiles

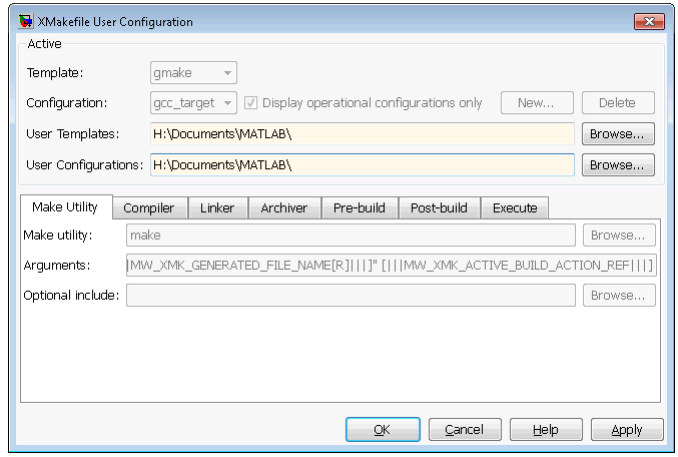
Update your model Configuration Parameters to use a makefile instead of an IDE when you build software from the model:

1. Configure your model for your IDE, tool chain, and target hardware, as described in "Configure Target Hardware Resources" on page 72-3.
2. In the Configuration Parameters dialog, under the **Code Generation** tab, select **Coder Target**.
3. Set **Build format** to `Makefile`. For more information, see "Build format" on page 72-7.
4. Set **Build action** to `Build_and_execute`. For more information, see "Build action" on page 72-7.

Choosing an XMakefile Configuration

Configure how to generate makefiles:

1. Enter `xmakefilesetup` on the MATLAB Command Window. The software opens an XMakefile User Configuration dialog.



2 Set the **Template** parameter to the option that matches the **Configuration** parameter.

> **Note:** In most cases, the only option for **Template** is `gmake`. However, if you have installed a Support Package, **Template** can have multiple options.

3 For **Configuration**, select the option that describes your software build toolchain and target platform. Click **Apply**.

Note: Changing some elements of the XMakefile dialog disables other elements until you apply the changes. Click **Apply** or **OK** after changing:

- **Template**
- **Configurations**
- **User Templates**
- **User Configurations**
- **Tool Directories**

Note: With the XMakefile User Configuration dialog, if you have an Embedded Coder license and do not have a Simulink Coder license, the **Configuration** list includes two unsupported options: `gcc_target` or `msvs_host`. Disregard those two configurations. Choose one of the other configurations.

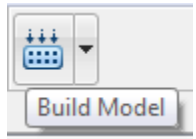
Things to consider while setting **Configuration**:

- Selecting **Display operational configurations only** hides configurations that contain incomplete or invalid information. For a configuration to be operational, the vendor tool chain must be installed, and the configuration must have the valid paths for each component of the vendor tool chain. For more information, see "Making an XMakefile Configuration Operational" on page 72-16.
- To display the configurations, including non-operational configurations, clear **Display operational configurations only**.
- The list of configurations can include non-editable configurations defined in the software and editable configurations defined by you.
- To create a new editable configuration, use the **New** button.

- For more information, see “XMakefile User Configuration dialog” on page 72-22.

Building Your Model

In your model, click **Build Model**.



This action creates a makefile and performs the other actions you specified in **Build action**.

By default, this process outputs files in the `<builddir>/<buildconfiguration>` folder. For example, in `model_name/CustomMW`.

Making an XMakefile Configuration Operational

When the XMakefile utility starts, it checks each configuration file to verify that the specified paths for the vendor tool chain are valid. If the paths are not valid, the configuration is non-operational. Typically, the cause of this problem is a difference between the path in the configuration and the actual path of the vendor toolchain.

To make a configuration operational:

1. Clear **Display operational configurations only** to display non-operational configurations.
2. Select the non-operational configuration from the **Configuration** options.
3. When you click **Apply**, a new dialog prompts you for the folder path of the missing resources the configuration requires.

   Use mapped network drives instead of UNC paths to specify directory locations. Using UNC paths with compilers that do not support them causes build errors.

Creating a New XMakefile Configuration

Overview

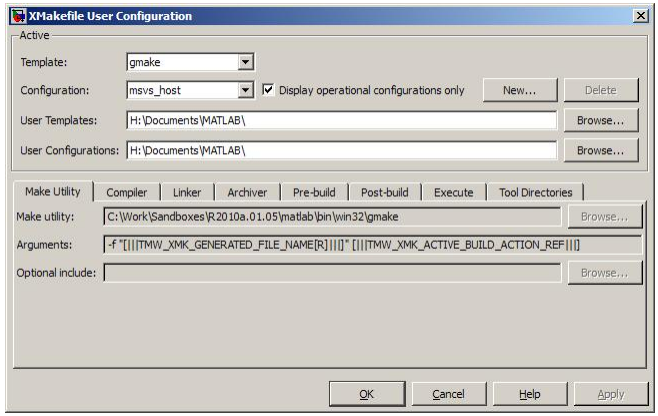
This example shows you how to add support for a software development toolchain to the XMakefile utility. This example uses the Intel Compiler and an IDE.

Note: To specify directory locations, use mapped network drives instead of UNC paths. UNC paths cause build errors with compilers that do not support them.

Create a Configuration

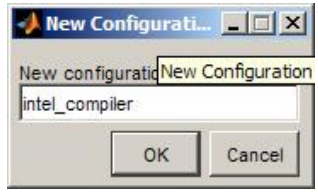
When you click **New**, the new configuration inherits values and behavior from the current configuration. To create a configuration for the Intel Compiler, clone a configuration from one of these configurations: `montavista_arm` and `gcc_target`.

Open the XMakefile User Configuration UI by typing `xmakefilesetup` at the MATLAB prompt. This action displays the following dialog.

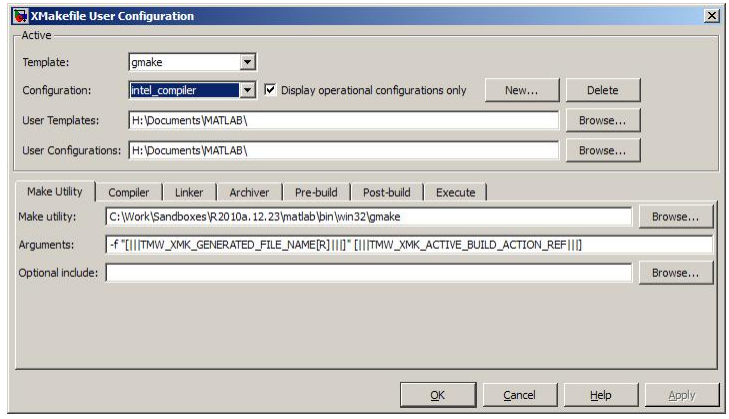


Select an existing configuration, such as `montavista_arm` or `gcc_target`. Click the **New** button.

A pop-up dialog prompts you for the name of the new configuration. Enter `intel_compiler` and click **OK**.



The dialog displays a new configuration called `intel_compiler`, based on the previous configuration.

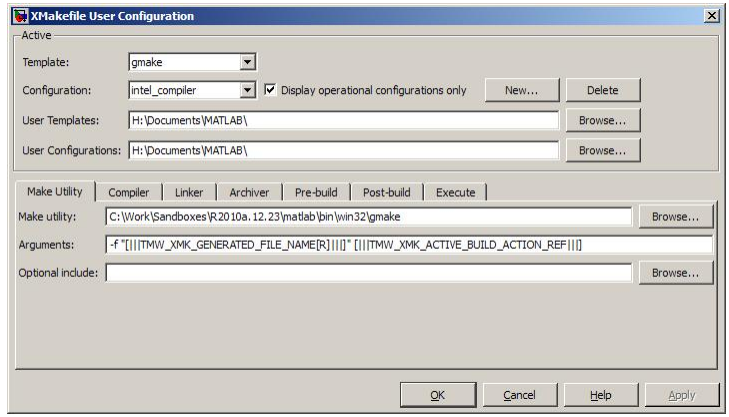


Modify the Configuration

Adjust the compiler, linker, and archiver settings of the newly created configuration. This example assumes the location of the Intel compiler is `C:\Program Files\Intel\Compiler\`.

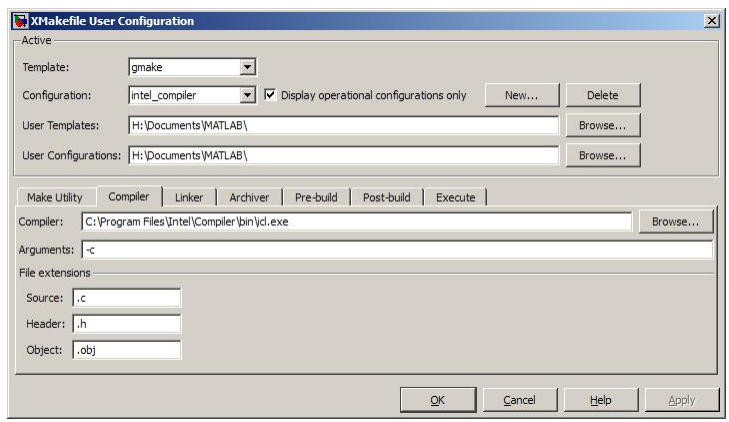
Make Utility

You do not need to make changes. This configuration uses the gmake tool that ships with MATLAB.



Compiler

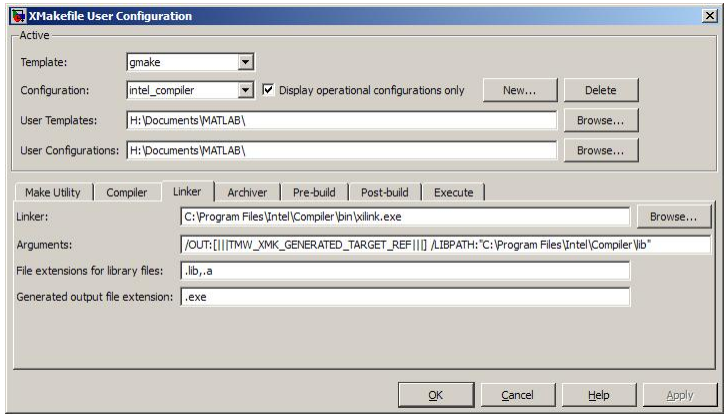
For **Compiler**, enter the location of `icl.exe` in the Intel installation.



Linker

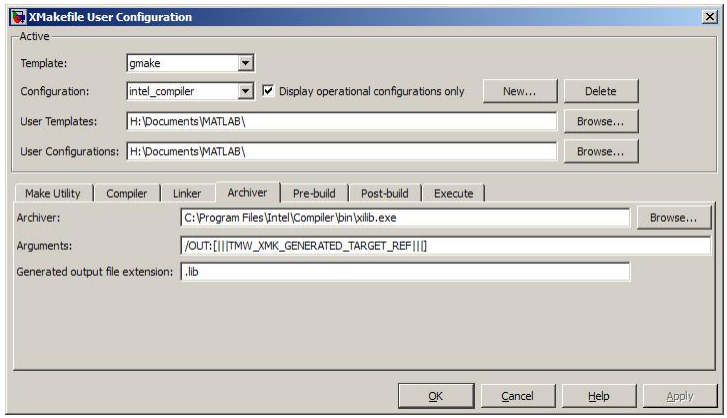
For **Linker**, enter the location of the linker executable, xilink.exe.

For **Arguments**, add the /LIBPATH path to the Intel libraries.



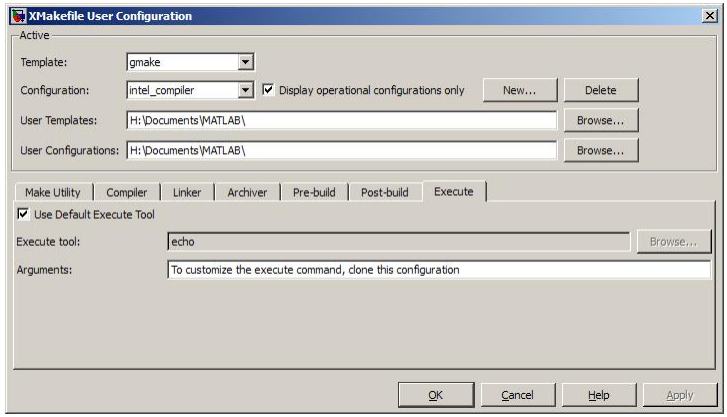
Archiver

For **Archiver**, enter the location of the archiver, `xilib.exe`. Confirm that **File extensions for library files** includes `.lib`.



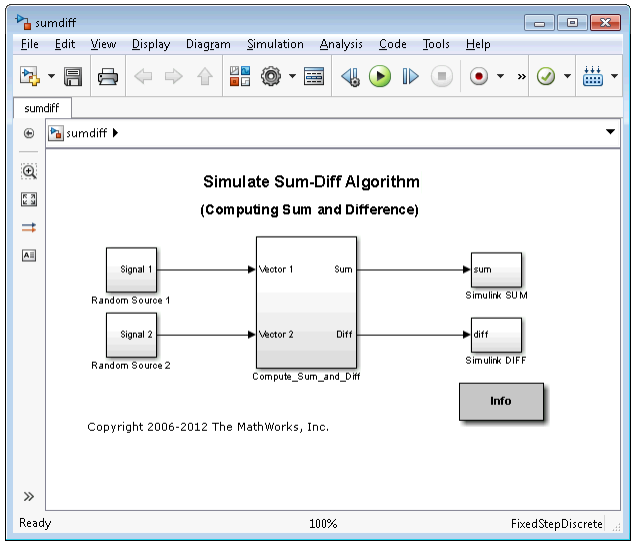
Other tabs

For this example, ignore the remaining tabs. In other circumstances, you can use them to configure additional build actions. In a later step of this example, you will configure the software to automatically build and run the generated code.



Test the Configuration

Open the "sumdiff" model by entering `sumdiff` on the MATLAB prompt.



Configure the `summdiff` model for use with an IDE. Follow the steps in "Configure Target Hardware Resources" on page 72-3, set the **IDE/Tool Chain** parameter, set **Board** to `Custom`, and **Processor** to `Intel x86/Pentium`.

On the Tool Chain Automation page, set **Operating System** to `None` or select `Windows`. Click **OK**.

Open the Configuration Parameters for the sumdiff model by pressing **Ctrl+E**. Set **Build format** to `Makefile` and **Build action** to `Build_and_execute`.

Save the model to a temporary location, such as `C:\Temp\IntelTest\`.

Set that location as a Current Folder by typing cd `C:\temp\IntelTest\` at the MATLAB prompt.

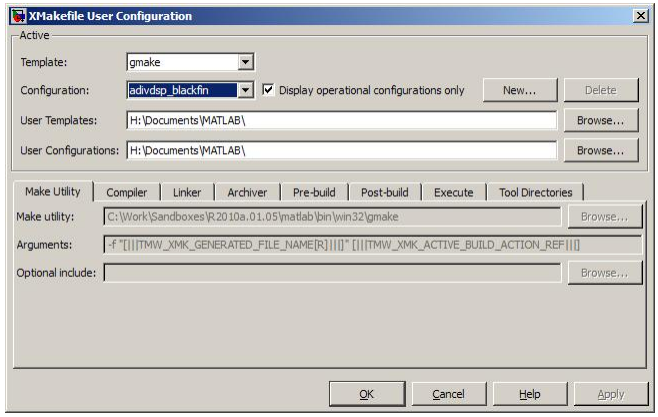
Build the model by pressing **Ctrl+B**. The MATLAB Command Window displays something like:

```
### TLC code generation complete.
### Creating HTML report file sumdiff_codegen_rpt.html
### Creating project: c:\temp\IntelTest\sumdiff_idenameide\sumdiff.mk
### Project creation done.
### Building project...
### Build done.
### Downloading program: c:\temp\IntelTest\sumdiff_idenameide\sumdiff
### Download done.
```

A command window comes up showing the running model. Terminate the generated executable by pressing **Ctrl+C**.

XMakefile User Configuration dialog

Active



Template

Set the **Template** parameter to the option that matches the **Configuration** parameter.

Note: In most cases, the only option for **Template** is `gmake`. However, if you have installed a Support Package, **Template** can have multiple options.

The template defines the syntax rules for writing the contents of the makefile or buildfile. The default template is `gmake`, which works with the GNU make utility.

To add templates to this parameter, save them as .mkt files to the location specified by the **User Templates** parameter. For more information, see “User Templates” on page 72-24.

Configuration

Select the configuration that best describes your toolchain and target hardware.

You cannot edit or delete the configurations provided by MathWorks. You can, however, edit and delete the configurations that you create.

Use the **New** button to create an editable copy of the currently selected configuration.

Use the **Delete** button to delete a configuration you created.

Note: You cannot edit or delete the configurations provided by MathWorks.

Note: Use mapped network drives instead of UNC paths to specify directory locations. Using UNC paths with compilers that do not support them causes build errors.

Display operational configurations only

When you open the XMakefile User Configuration dialog, the software verifies that each configuration provided by MathWorks contains valid paths to the executable files it uses. If the paths are valid, the configuration is operational. If the paths are not valid, the configuration is not operational.

This setting only applies to configurations provided by MathWorks, not configurations you create.

To display valid configurations, select **Display operational configurations only**.

To display the configurations, including non-operational configurations, clear **Display operational configurations only**.

For more information, see "Making an XMakefile Configuration Operational" on page 72-16.

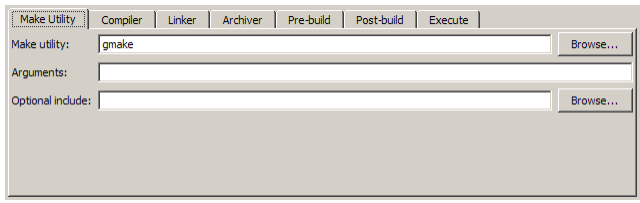
User Templates

Set the path of the folder to which you can add template files. Saving templates files with the .mkt extension to this folder adds them to the **Templates** options.

User Configurations

Set the location of configuration files you create with the **New** button.

Make Utility



Make utility

Set the path and filename of the make utility executable.

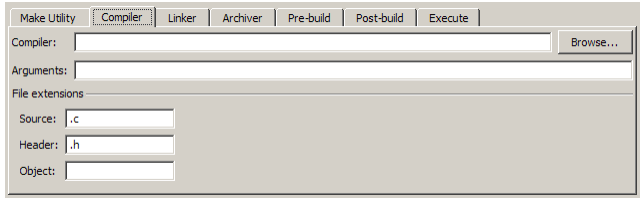
Arguments

Define the command-line arguments to pass to the make utility. For more information, consult the third-party documentation for your make utility.

Optional include

Set the path and file name of an optional makefile to include.

Compiler



Compiler

Set the path and file name of the compiler executable.

Arguments

Define the command-line arguments to pass to the compiler. For more information, consult the third-party documentation for your compiler.

Source

Define the file name extension for the source files. Use commas to separate multiple file extensions.

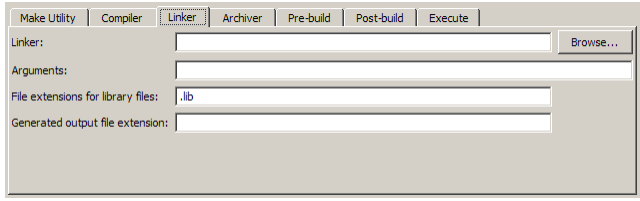
Header

Define the file name extension for the header files. Use commas to separate multiple file extensions.

Object

Define the file name extension for the object files.

Linker



Linker

Set the path and file name of the linker executable.

Arguments

Define the command-line arguments to pass to the linker. For more information, consult the third-party documentation for your linker.

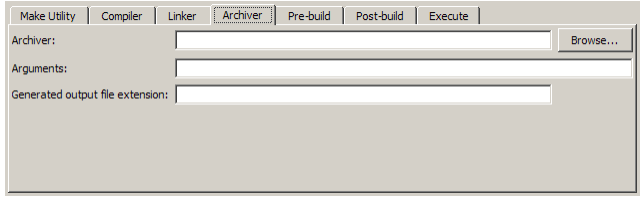
File extensions for library files

Define the file name extension for the file library files. Use commas to separate multiple file extensions.

Generated output file extension

Define the file name extension for the generated libraries or executables.

Archiver



Archiver

Set the path and file name of the archiver executable.

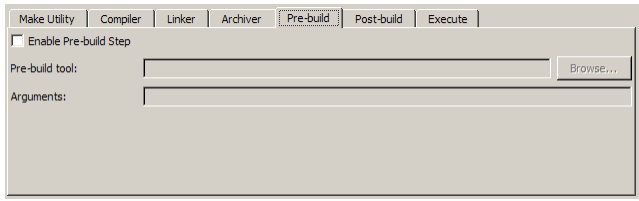
Arguments

Define the command-line arguments to pass to the archiver. For more information, consult the third-party documentation for your archiver.

Generated output file extension

Define the file name extension for the generated libraries.

Pre-build



Enable Prebuild Step

Select this check box to define a prebuild tool that runs before the compiler.

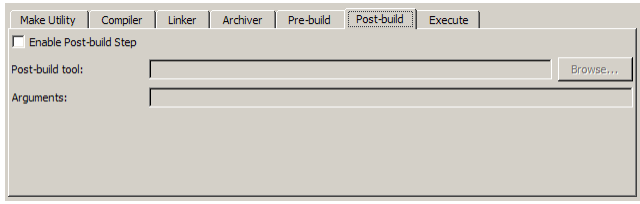
Prebuild tool

Set the path and file name of the prebuild tool executable.

Arguments

Define the command-line arguments to pass to the prebuild tool. For more information, consult the third-party documentation for your prebuild tool.

Post-build



Enable Postbuild Step

Select this check box to define a postbuild tool that runs after the compiler or linker.

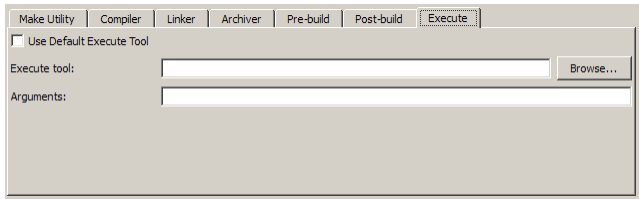
Postbuild tool

Set the path and file name of the postbuild tool executable.

Arguments

Define the command-line arguments to pass to the postbuild tool. For more information, consult the third-party documentation for your postbuild tool.

Execute



Use Default Execute Tool

Select this check box to use the generated derivative as the execute tool when the build process is complete. Uncheck it to specify a different tool. The default value, `echo`, simply displays a message that the build process is complete.

Note: On the Linux operating system, multirate multitasking executables require root privileges to schedule POSIX threads with real-time priority. If you are using makefiles to build multirate multitasking executables on your Linux development system, you cannot use **Execute tool** to run the executable. Instead, use the Linux command, `sudo`, to run the executable.

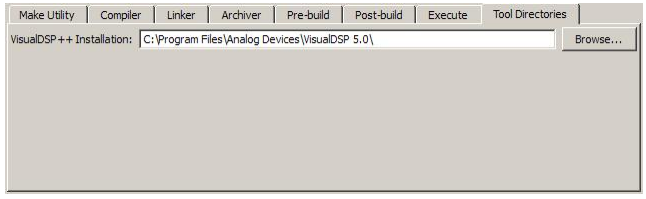
Execute tool

Set the path and file name of the execute tool executable or built-in command.

Arguments

Define the command-line arguments to pass to the execute tool. For more information, consult the third-party documentation for your execute tool.

Tool Directories



Installation

Use the Tool Directories tab to change the toolchain path of an operational configuration.

For example, if you installed two versions of a vendor build tool in separate folders, you can use the **Installation** path to change which one the configuration uses.

Verification and Profiling Generated Code in Embedded Coder

PIL Simulation for IDE and Toolchain Targets

In this section...

Overview

Verification consists broadly of running generated code on a processor and verifying that the code does what you intend. Embedded Coder provides processor-in-the-loop (PIL) simulation to meet this need. PIL compares the numeric output of your model under simulation with the numeric output of your model running as an executable on a target hardware.

With PIL, you run your generated code on a target hardware or instruction set simulator. To verify your generated code, you compare the output of model simulation modes, such as Normal or Accelerator, with the output of the generated code running on the processor. You can switch between simulation and PIL modes. This flexibility allows you to verify the generated code by executing the model as compiled code in the target environment. You can model and test your embedded software component in Simulink and then reuse your regression test suites across simulation and compiled object code. This process avoids the time-consuming process of leaving the Simulink software environment to run tests again on object code compiled for the production hardware.

Embedded Coder supports the following PIL approaches:

- Model block PIL
- Top-model PIL
- PIL block

When you use makefiles with PIL, use the "model block PIL" approach. With makefiles, the other two approaches, "top-model PIL" and "PIL block", and are not supported.

PIL Approaches

Model Block PIL

Use model block PIL to:

- Verify code generated for referenced models (model reference code interface).
- Provide a test harness model (or a system model) to generate test vector or stimulus inputs.
- Switch a model block between normal, SIL, or PIL simulation modes.

To perform a model block PIL simulation, start with a top-model that contains a model block. The top-model serves as a test harness, providing inputs and outputs for the model block. The model block references the model you plan to run on target hardware. During PIL simulation, the referenced model runs on the target hardware.

For more information about using the model block, see Model, Model Variants (Simulink) and "Model Referencing" (Simulink).

By default, your MathWorks software uses the IDE debugger for PIL communications with the target hardware. To achieve faster communications, consider using one of the alternatives presented in "Communications" on page 73-7.

To use model block PIL:

1. Create and share a configuration reference between the top model and the referenced model, as described in "Share a Configuration for Multiple Models" (Simulink).
2. Right-click the Model block, and select **ModelReference Parameters**.
3. When the software displays the **Function Block Parameters: Model** dialog box, set **Simulation mode** to `Processor-in-the-loop (PIL)` and click **OK**.
4. Open the model block.
5. In the referenced model (model block) Configuration Parameters (**Ctrl+E**), under **Code Generation** > **Coder Target**, set **Build action** set to `Archive_library`. This action avoids a warning when you start the simulation.
6. Save the changes to both models.

7 In the top-model menu bar, select **Simulation** > **Run**. This action builds the referenced model in the model block, downloads it to your target hardware, and runs the PIL simulation.

Note: In the top-model Configuration Parameters (**Ctrl+E**), under **Code Generation** > **Coder Target**, leave **Build action** set to `Build_and_execute`. Do not change **Build action** to `Create_Processor_In_the_Loop_Project`.

Top-Model PIL

Use top-model PIL to:

- Verify code generated for a top-model (standalone code interface).
- Load test vectors or stimulus inputs from the MATLAB workspace.
- Switch the entire model between normal and SIL or PIL simulation modes.

Setting Model Configuration Parameters to Generate the PIL Application

Configure your model to generate the PIL executable from your model:

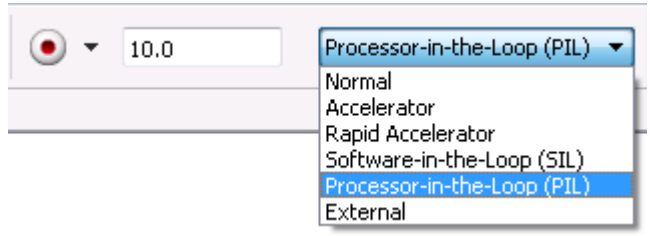
1 Configure your model to run on target hardware, as described in "Configure Target Hardware Resources" on page 72-3.
2 From the model toolstrip, select **Simulation** > **Model Configuration Parameters**.
3 In Configuration Parameters, select **Code Generation**.
4 Set **System Target File** to `idelink_ert.tlc`.
5 From the list of panes under **Code Generation**, choose **Coder Target**.
6 Set **Build format** to `Project`.
7 Set **Build action** to `Create_processor_in_the_loop_project`.
8 Click **OK** to close the Configuration Parameters dialog box.

For more information, see "Code Generation: Coder Target Pane".

Running the Top-Model PIL Application

To create a PIL block, perform the following steps:

1 In the model toolstrip, set the Simulation mode to `Processor-in-the-loop`.



2 In the model toolstrip, click Run.

A new Simulink Editor opens with the new PIL model block in it. The third-party IDE compiles and links the PIL executable file. Follow the progress of the build process in the MATLAB Command Window.

PIL Block

Use the PIL block to:

- Verify code generated for a top-model (standalone code interface) or subsystem (right-click build standalone code interface).
- Represent a component running in SIL or PIL mode. The test harness model or a system model provides test vector or stimulus inputs.

Preparing Your Model to Generate a PIL Block

Start with a model that contains the algorithm blocks you want to verify on the processor as compiled object code. To create a PIL application and PIL block from your algorithm subsystem, follow these steps:

1 Identify the algorithm blocks to cosimulate.

2 Convert those blocks into an unmasked subsystem in your model.

For information about how to convert your process to a subsystem, refer to Creating Subsystems (Simulink) in *Using Simulink* or in the online Help system.

3 Open the newly created subsystem.

4 Configure your subsystem to run on target hardware, as described in “Configure Target Hardware Resources” on page 72-3.

Setting Model Configuration Parameters to Generate the PIL Application

After you create your subsystem, set the Configuration Parameters for your model to enable the model to generate a PIL block.

Configure your model to enable it to generate PIL algorithm code and a PIL block from your subsystem:

1. From the model menu bar, select **Simulation > Model Configuration Parameters**. This action opens the Configuration Parameters dialog box.
2. In the Configuration Parameters dialog box, select **Code Generation**.
3. Set **System Target File** to `idelink_ert.tlc`.
4. From the list of panes under **Code Generation**, choose **Coder Target**.
5. Set **Build format** to `Project`.
6. Set **Build action** to `Create_processor_in_the_loop_project`.
7. Click **OK** to close the Configuration Parameters dialog box.

For more information, see "Code Generation: Coder Target Pane".

Creating the PIL Block from a Subsystem

To create a PIL block, perform the following steps:

1. Right-click the masked subsystem in your model and select **C/C++ Code > Build This Subsystem** from the context menu.

   A new Simulink Editor opens and the new PIL block appears in it. The third-party IDE compiles and links the PIL executable file.

   This step builds the PIL algorithm object code and a PIL block that corresponds to the subsystem, with the same inputs and outputs. Follow the progress of the build process in the MATLAB Command Window.
2. Copy the new PIL block from the new model to your model. To simulate the subsystem processes concurrently, place it parallel to your masked subsystem. Otherwise, replace the subsystem with the PIL block.

   To see a PIL block in a parallel masked subsystem, search the product help for *Getting Started with Application Development* and select the example that matches your IDE.

Note: Models can have multiple PIL blocks for different subsystems. They cannot have more than one PIL block for the same subsystem. Including multiple PIL blocks for the same subsystem causes errors and inaccurate results.

Communications

Choose one of the following communication methods for transferring code and data during PIL simulations:

Method	Speed	Comments
IDE Debugger	Slow	• Supports PIL communications with an executable running an embedded target hardware. • Supports the largest number of targets. • Requires a physical connection between host and target hardware. • Only works with builds from IDE projects. Does not work with builds from makefiles.
TCP/IP	Fast	• Supports PIL communications with an executable running on a Linux or Windows host. • Supports embedded targets running Linux, TI DSP/BIOS, and Wind River VxWorks. • Requires network connection between host and target hardware. • Works with builds from IDE projects and from makefiles.

TCP/IP

You can use TCP/IP for PIL communications with target hardware running:

- Linux

- Wind River VxWorks

Using TCP/IP for PIL communications is typically faster than using a debugger, particularly for large data sets, such as with video and audio applications.

It also works well when you build an application on a remote Linux target using the `remoteBuild` function.

You can use TCP/IP with the following PIL approaches:

- Top-model PIL
- Model block PIL

TCP/IP does not work with the Subsystem PIL approach.

To enable and configure TCP/IP with PIL:

1. Set up a PIL simulation according to the PIL approach you have chosen.
2. In the MATLAB Command Window, use `setpref` to specify the IP address of the PIL server (`servername`).

   If you are running the PIL server on a remote target, specify the IP address of the target hardware. For example:

   ```
   setpref('MathWorks_Embedded_IDE_Link_PIL_Preferences','servername','144.212.109.114');
   ```

   If you are running PIL server locally, on your host Windows or Linux system, enter `'localhost'` instead of an IP address:

   ```
   setpref('MathWorks_Embedded_IDE_Link_PIL_Preferences','servername','localhost');
   ```
3. Specify the TCP/IP port number to use for PIL data communication. Use one of the free ports in your system. For example:

   ```
   setpref('MathWorks_Embedded_IDE_Link_PIL_Preferences','portnum', 17025);
   ```
4. Enable PIL communications over TCP/IP:

   ```
   setpref('MathWorks_Embedded_IDE_Link_PIL_Preferences','enabletcpip', true);
   ```

   To disable PIL communications over TCP/IP, change the value to `false`. This action automatically enables PIL communications over an IDE debugger, if an IDE is available.
5. Open the Configuration Parameters in your model. On the Coder Target pane, set the **Operating System** parameter to the operating system your target hardware is running.

Note: You cannot use TCP/IP for PIL when the value of **Operating System** is `None`.

6 Regenerate the code or PIL block.

To disable PIL communications over TCP/IP, enter:

```
setpref('MathWorks_Embedded_IDE_Link_PIL_Preferences','enabletcpip', false);
```

IDE Debugger

To enable PIL communications over an IDE debugger, disable PIL communications over TCP/IP and SCI by entering the following commands:

```
setpref('MathWorks_Embedded_IDE_Link_PIL_Preferences','enabletcpip',false);
setpref('MathWorks_Embedded_IDE_Link_PIL_Preferences','enableserial',false);
```

Then regenerate the code or PIL block.

Using IDE debugger for PIL communication only works when you build your code from IDE projects. Using IDE debugger for PIL communication does not work with builds from makefiles.

Configuring Breakpoints

You can use the `setStartApplicationPause` API to set breakpoints in the PIL application on the *first* PIL block simulation. If you do not use the `setStartApplicationPause` API, you can configure breakpoints after the initial run. The breakpoints remain active for subsequent runs.

You can enter the following static API method to pause after loading the application and manually configure breakpoints:

```
rtw.connectivity.Launcher.setStartApplicationPause(pauseAmount)
```

About this method:

- This method tells the MATLAB session to pause immediately after the PIL launcher starts the PIL application.
- `pauseAmount` is a pause time in seconds. To disable the pause, enter 0.

When you do not specify a pause, the software displays the following message:

```
### To pause during PIL application start, run: >> rtw.connectivity.Launcher.
setStartApplicationPause(120)
```

The default pause is 120 sec. You can change this value.

When you specify a pause, a Start PIL Application Pause message box appears and displays following message:

```
Pausing during PIL application start for 120s (click OK to continue).
To disable this pause, see the hyperlink in the MATLAB command window.
```

- The MATLAB Command Window shows the following text:

  ```
  ### To remove the pause during PIL application start,
  run: >> rtw.connectivity.Launcher. setStartApplicationPause(0)
  ```
  where `rtw.connectivity.Launcher. setStartApplicationPause(0)` is a hyperlink.
- The pause times out, or you can clear it early by closing the message box.
- During the pause, you cannot access MATLAB and thus cannot configure breakpoints programmatically via the IDE Automation Interface API.
- For the PIL block, the debugger stays open between simulation runs. When you perform an initial simulation run, you can automatically configure breakpoints via the IDE Automation Interface API before starting the second simulation.

Running Your PIL Application to Perform Simulation and Verification

After you add PIL block to your model, add the required pause in seconds, using the following command in the MATLAB command prompt:

```
rtw.connectivity.Launcher.setStartApplicationPause(120)
```
Then click **Simulation** > **Run** or press **Ctrl+T** to run the PIL simulation and view the results.

Note The pause command is to make sure that the automatic download of PIL completes, before the model starts executing.

Definitions

PIL Algorithm

The algorithmic code, which corresponds to a subsystem or portion of a model, to test during the PIL simulation. The PIL algorithm is in compiled object form to enable verification at the object level.

PIL Application

The executable application that runs on the processor platform. The code generator produces code for a PIL application by augmenting your algorithmic code with the PIL execution framework. The PIL execution framework code compiles as part of your embedded application.

The PIL execution framework code includes the `string.h` header file so that the PIL application can use the `memcpy` function. The PIL application uses `memcpy` to exchange data between the Simulink model and the simulation processor.

PIL Block

When you build a subsystem from a model for PIL, the process creates a PIL block optimized for PIL simulation. When you run the simulation, the PIL block acts as the interface between the model and the PIL application running on the processor. The PIL block inherits the signal names and shape from the source subsystem in your model. Inheritance is convenient for copying the PIL block into the model to replace the original subsystem for simulation.

PIL Issues and Limitations

Consider the following issues when you work with PIL blocks.

Constraints

When using PIL in your models, keep in mind the following constraints:

- Models can have multiple PIL blocks for different subsystems. They cannot have more than one PIL block for the same subsystem. Including multiple PIL blocks for the same subsystem causes errors and inaccurate results.
- A model can contain a single model block running PIL mode.
- A model can contain a subsystem PIL block or a model block in PIL mode, but not both.

Generic PIL Issues

Refer to PIL Feature Support and Limitations on page 64-61 for general information about using the PIL block with embedded link products.

Simulink Coder grt.tlc-Based Targets Not Supported

PIL does not support `grt.tlc` system target files.

To use PIL, set **System target file** in the Configuration Parameters > Code Generation pane to `idelink_ert.tlc`.

Code Execution Profiling for IDE and Toolchain Targets

In this section...

Execution-Time Profiling

You can measure the execution times during a standalone execution or processor-in-the-loop (PIL) simulation. You can generate execution time profiles for synchronous tasks, asynchronous tasks, and atomic subsystems. Use this feature to check whether your code runs in real time on your target hardware. For details, see "Perform Execution-Time Profiling for IDE and Toolchain Targets" on page 73-16.

You can use this profiling for generated code in the following cases:

- **Code Generation** > **System target file** is `ert.tlc` and **Code Generation** > **Target hardware** is not `None`, for example, `ARM Cortex-A9 (QEMU)` or `ARM Cortex-M3 (QEMU)`.
- **Code Generation** > **System target file** is `idelink_ert.tlc`

The following table provides execution time profiling support information.

Mode	Coder Target > Tool Chain Automation > Build format parameter value
Standalone execution or PIL simulation	`Project`
PIL simulation	`Makefile`

Stack Profiling

With stack profiling, you can determine how generated code uses the processor system stack. Using the `profile` method, you can initialize and test the size and usage of the stack. See "Perform Stack Profiling with IDE and Toolchain Targets" on page 73-22. This information can help you optimize both the size of the stack and how your code uses the stack.

You can use this profiling for generated code in the following cases:

- **Code Generation** > **System target file** is `ert.tlc` and **Code Generation** > **Target hardware** is not `None`, for example, `ARM Cortex–A9 (QEMU)` or `ARM Cortex–M3 (QEMU)`.
- **Code Generation** > **System target file** is `idelink_ert.tlc`

Note: Stack profiling is not supported on embedded targets that run an operating system or RTOS.

To provide stack profiling, `profile` writes a known pattern to the addresses in the stack. After you run your application for a while, and then stop your application, `profile` examines the contents of the stack addresses. `profile` counts each address that does not contain the known pattern. The total number of addresses that have been used, compared to the total number of addresses that you allocated, becomes the stack usage profile. This profile process does not determine how often your application changes an address.

You can profile the stack with the manually written code in a project and the code that you generate from a model.

When you use `profile` to initialize and test the stack operation, the software returns a report that contains information about stack size, usage, addresses, and direction. With this information, you can modify your code to use the stack efficiently. The following program listing shows the stack usage results from running an application on a simulator.

```
profile(IDE_Obj,'stack','report')

Maximum stack usage:

System Stack: 532/1024 (51.95%) MAUs used.


           name: System Stack
   startAddress: [512    0]
     endAddress: [1535     0]
      stackSize: 1024 MAUs
growthDirection: ascending
```

The following table describes the entries in the report.

Report Entry	Units	Description
`System Stack`	Minimum Addressable Unit (MAU)	Maximum number of MAUs used and the total MAUs allocated for the stack.
`name`	Character vector for the stack name	Lists the name assigned to the stack.
`startAddress`	Decimal address and page	Lists the address of the stack start and the memory page.
`endAddress`	Decimal address and page	Lists the address of the end of the stack and the memory page.
`stackSize`	Addresses	Reports number of address locations, in MAUs, allocated for the stack.
`growthDirection`	Not applicable	Reports whether the stack grows from the lower address to the higher address (`ascending`) or from higher to lower (`descending`).

Related Examples

- "Perform Execution-Time Profiling for IDE and Toolchain Targets" on page 73-16
- "Perform Stack Profiling with IDE and Toolchain Targets" on page 73-22

Perform Execution-Time Profiling for IDE and Toolchain Targets

In this section...

Execution-Time Profiling During Standalone Execution

During standalone execution, instrumentation in the generated code collects execution-time samples, which are stored in target hardware memory. After halting target hardware execution, you can use the `profile` function to transfer the execution data from target hardware memory to the MATLAB workspace for viewing and analysis.

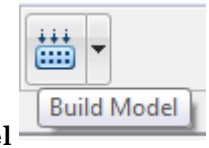
You can perform profiling by task or subsystem. A profiling sample represents a task or subsystem execution instance. Each sample requires two memory locations, one for the start time and one for the end time. Therefore, you must specify a buffer size that is twice the number of required profiling samples. Sample collection begins with the start of code execution and ends when the buffer is full.

Task Profiling

To configure a model for task execution profiling:

1. In your model, select **Simulation** > **Model Configuration Parameters**.
2. Select the **Code Generation** > **Coder Target** pane.
3. Set **Build format** to `Project` and set **Build action** to `Build_and_execute`.
4. Select **Profile real-time execution**.
5. In the **Profile by** list, select **Tasks**.
6. Specify **Number of profiling samples to collect**, the size of the buffer that stores execution data. Enter a value that is twice the number of profiling samples you require.
7. Click **OK**.

To view the execution profile for your model:

1 Click **Build Model** 

on the model toolstrip. This action builds, loads, and runs your code on the processor.

2 To stop the running program, select **Debug** > **Halt** in the IDE or use `IDE_obj.halt` from the MATLAB command line. Gathering profiling data from a running program can yield inaccurate results.

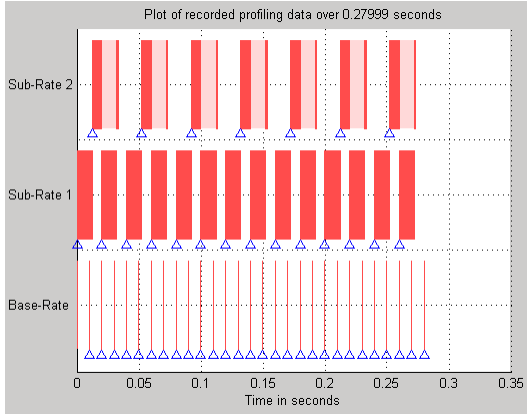
3 At the MATLAB command prompt, enter

```
profile(IDE_Obj,'execution','report')
```

to view the MATLAB software graphic of the execution report and the HTML execution report.

For more information about other reporting options, see the product help for the `profile` function.

The following profiling plot is from an application that runs with three rates — the base rate and two slower rates. Gaps in the `Sub-Rate 2` task bars indicate preempted operations.



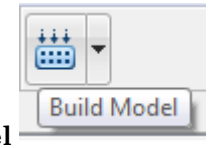
Subsystem Profiling

To configure a model for subsystem execution profiling:

1 Configure your model for your IDE, tool chain, and target hardware, as described in “Configure Target Hardware Resources” on page 72-3.

2. On the **Coder Target** pane, set **Build format** to `Project` and set **Build action** to `Build_and_execute`.
3. Select **Profile real-time execution**.
4. In the **Profile by** list, select `Atomic subsystems`.
5. Specify **Number of profiling samples to collect**, the size of the buffer that stores execution data. Enter a value that is twice the number of profiling samples you require.
6. Click **OK**.

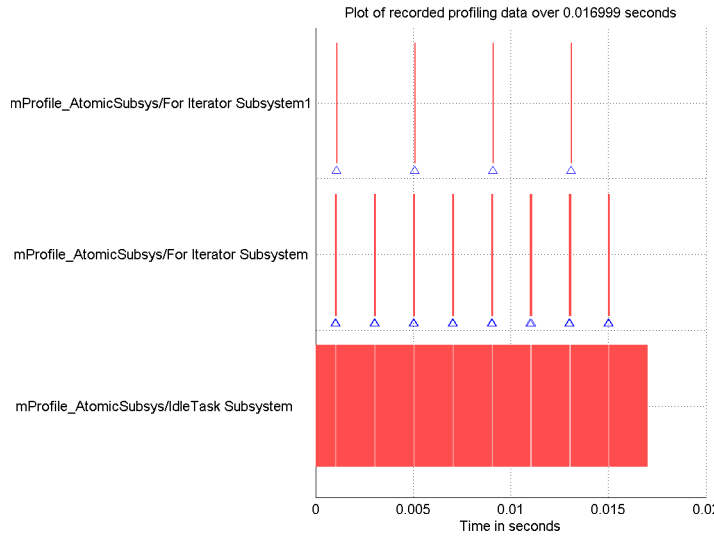
To view the execution profile for your model:

1. Click **Build Model** 

on the model toolstrip. This action builds, loads, and runs your code on the processor.
2. To stop the running program, select **Debug** > **Halt** in the IDE or use `IDE_obj.halt` from the MATLAB command line. Gathering profiling data from a running program can yield inaccurate results.
3. At the MATLAB command prompt, enter:

   ```
   profile(IDE_Obj, 'execution','report')
   ```
   to view the MATLAB software graphic of the execution report and the HTML execution report.

The following profiling plot is from an application with three subsystems — `For Iterator Subsystem`, `For Iterator Subsystem1`, and `Idle Task Subsystem`.



Execution-Time Profiling During PIL Simulation

During a processor-in-the-loop (PIL) simulation, you can profile execution times of synchronous tasks. The software stores the profile data in a `coder.profile.ExecutionTime` object, located in the MATLAB workspace. After halting the PIL simulation, you can view and analyze the data.

Gathering Execution Profile Data

1. Configure a model for PIL simulation, as described in "PIL Simulation for IDE and Toolchain Targets" on page 73-2.
2. In your model, select **Simulation** > **Model Configuration Parameters**.
3. In the Configuration Parameters dialog box, select **Code Generation**, and then **Verification**.
4. Select the **Measure task execution time** check box.
5. Provide a valid MATLAB variable name in the **Workspace** edit box. The software uses this name when it creates the `coder.profile.ExecutionTime` object.
6. Click **OK** to close the Configuration Parameters dialog box.

7 Run the PIL simulation, as described in "PIL Simulation for IDE and Toolchain Targets" on page 73-2.

   The software creates the `coder.profile.ExecutionTime` object and stores the execution profile data in it.

8 Halt the PIL simulation.

Analyzing the Execution Profile Data

After halting the PIL simulation, you can view or analyze the data in the `coder.profile.ExecutionTime` object. For more information, see:

- "View and Compare Code Execution Times" on page 58-7
- "Analyze Code Execution Data" on page 58-18

Depending on the target, the execution profile data is measured in seconds or timer ticks.

Targets	Units
Analog Devices Blackfin, SHARC, and TigerSHARC processors with VisualDSP++ IDE	Timer Ticks
ARM processors running Wind River VxWorks	Timer Ticks

The `coder.profile.ExecutionTime` class has property `TimerTicksPerSecond` for getting and setting the data units. You can use this property on the execution profile data object after halting the PIL simulation. When the data unit is timer ticks, using the `TimerTicksPerSecond` property converts the data units to seconds.

See Also

`profile` | `TimerTicksPerSecond`

Related Examples

- "Configure Target Hardware Resources" on page 72-3
- "PIL Simulation for IDE and Toolchain Targets" on page 73-2
- "View and Compare Code Execution Times" on page 58-7

- “Analyze Code Execution Data” on page 58-18

Perform Stack Profiling with IDE and Toolchain Targets

To profile the system stack operation:

1. Load an application.
2. Set up the stack to enable profiling.
3. Run your application.
4. Request the stack profile information.

Follow these steps to profile the stack as your application interacts with it. This particular example uses Texas Instruments Code Composer Studio 3.3. However, you can generalize from this example to another supported IDE.

1. Load the application to profile.
2. Use the `profile` method with the **`setup`** input keyword to initialize the stack to a known state.

   ```
   profile(IDE_Obj,'stack','setup')
   ```

   With the **`setup`** input argument, `profile` writes a known pattern into the addresses that compose the stack.
3. Run your application.
4. Stop your running application. Stack use results gathered from an application that is running may be inaccurate.
5. Use the `profile` method to capture and view the results of profiling the stack.

   ```
   profile(IDE_Obj,'stack','report')
   ```

The following example shows how to set up and profile the stack. The IDE link handle object, `IDE_Obj`, must exist in your MATLAB workspace and your application must be loaded on your processor. This example comes from a TI C6713 simulator.

```
profile(IDE_Obj,'stack','setup')  % Set up processor stack
%by write A5 to the stack addresses.

Maximum stack usage:

System Stack: 0/1024 (0%) MAUs used.


           name: System Stack
   startAddress: [512    0]
     endAddress: [1535     0]
      stackSize: 1024 MAUs
```

```
growthDirection: ascending


run(IDE_Obj)
halt(IDE_Obj)
profile(IDE_Obj,'stack','report') % Request stack use report.

Maximum stack usage:

System Stack: 356/1024 (34.77%) MAUs used.


          name: System Stack
   startAddress: [512    0]
     endAddress: [1535     0]
      stackSize: 1024 MAUs
growthDirection: ascending
```

Related Examples

- “Code Execution Profiling for IDE and Toolchain Targets” on page 73-13

74

Processor-Specific Optimizations for Embedded Targets in Embedded Coder

Replace Code for Embedded Targets

In this section...

Using a Processor-Specific Code Replacement Library to Optimize Code

You can optimize the code the code generator produces for a specific processor by configuring the code generator to use a code replacement library (CRL) during code generation. If you have an Embedded Coder license, you can develop and apply custom code replacement libraries.

For more information about replacing code, using code replacement libraries that MathWorks provides, see "What Is Code Replacement?" on page 38-2 and "Replace Code Generated from Simulink Models" on page 38-11. For information about developing code replacement libraries, see "What Is Code Replacement Customization?" on page 51-3 and "Develop a Code Replacement Library" on page 51-27.

Process of Determining Optimization Effects Using Real-Time Profiling Capability

You can use the real-time profiling capability to examine the results of applying the processor-specific library functions and operators to your generated code. After you select a processor-specific code replacement library, use the real-time execution profiling capability to examine the change in program execution time.

Use the following process to evaluate the effects of applying a processor-specific code replacement library when you generate code:

1. Enable real-time profiling in your model. Refer to "Code Execution Profiling".
2. Generate code for your project without specifying a code replacement library (the default **Code replacement library** setting is `None`).
3. Profile the code, and save the report.
4. Rebuild your project using a processor-specific code replacement library.

5 Profile the code, and save the second report.

6 Compare the profile report from running your application with the processor-specific library selected to the profile results in the first report with no code replacement library selected.

For an example of verifying the code optimization, search help for "Optimizing Embedded Code via Code Replacement Library" and select the example that matches your IDE.

Code Generation from MATLAB Code

Build Configuration for Code Generation from MATLAB Code

Specify Comment Style for C/C++ Code

In this section...

If you have an Embedded Coder , you can specify the comment style for C/C++ code generated from MATLAB code. Specify single-line style to generate single-line comments preceded by `//`. Specify multiline style to generate single-line or multiline comments delimited by `/*` and `*/`. Single-line style is the default for C++ code generation. Multiline style is the default for C code generation. For C code generation, select single-line comment style only if your compiler supports it.

Specify Comment Style Using the MATLAB Coder App

1. On the **Generate Code** page, to open the **Generate** dialog box, click the **Generate** arrow  .
2. Set **Build type** to one of the following:
   - `Source Code`
   - `Static Library (.lib)`
   - `Dynamic Library (.dll)`
   - `Executable (.exe)`
3. Click **More Settings**.
4. On the **Code Appearance** tab, select the **Include comments** check box if it is not already selected. By default, the **Include comments** check box is selected.
5. Set **Comment Style** to one of the following options.

Value	Description
`Auto(Use standard comment style of the target language)`	For C, generate multiline comments. For C++, generate single-line comments. (default)
`Single-line (Use C++-style comments)`	Generate single-line comments preceded by `//`.
`Multi-line (Use C-style comments)`	Generate single-line or multiline comments delimited by `/*` and `*/`.

Specify Comment Style Using the Command-Line Interface

1. Create a code configuration object for C/C++ code generation. For example, create a configuration object for C/C++ static library generation:

   ```
   cfg = coder.config('lib','ecoder',true);
   ```

2. Set the `CommentStyle` property to one of the following values:

Value	Description
`'Auto'`	For C, generate multiline comments. For C++, generate single-line comments. (default)
`'Single-line'`	Generate single-line comments preceded by `//`.
`'Multi-line'`	Generate single-line or multiline comments delimited by `/*` and `*/`.

   For example, this code sets the comment style to single-line style:

   ```
   cfg.CommentStyle='Single-line';
   ```

See Also

coder.EmbeddedCodeConfig

More About

- “Configure Build Settings” (MATLAB Coder)

Specify Indent Style for C/C++ Code

In this section...

If you have an Embedded Coder license, you can control the indent style and indent size in C/C++ code generated from MATLAB code. Indent style controls the placement of braces. Indent size controls the number of characters per indentation level.

You can specify the K&R indent style or the Allman indent style. Both styles:

- Place the function opening and closing braces on their own lines at the same indentation level as the function header.
- Indent code within the function according to the indent size.
- For blocks within a function, place closing braces on a new line at the same indentation level as the control statement.
- Indent code within a block according to the indent size.

The K&R style and the Allman style differ in their placement of the opening brace for a control statement. If you want the opening brace on the same line as its control statement, select the K&R style. Here is code that has the K&R indent style:

```
void addone(const double x[6], double z[6])
{
  int i0;
  for (i0 = 0; i0 < 6; i0++) {
    z[i0] = x[i0] + 1.0;
  }
}
```

If you want the opening brace on its own line, select the Allman style. Here is code that has the Allman indent style:

```
void addone(const double x[6], double z[6])
{
  int i0;
  for (i0 = 0; i0 < 6; i0++)
  {
    z[i0] = x[i0] + 1.0;
```

```
  }
}
```

Specify Indent Style Using the MATLAB Coder App

1. On the **Generate Code** page, to open the **Generate** dialog box, click the **Generate** arrow  .
2. Set **Build type** to one of the following:
   - `Source Code`
   - `Static Library (.lib)`
   - `Dynamic Library (.dll)`
   - `Executable (.exe)`
3. Click **More Settings**.
4. On the **All Settings** tab, under **Advanced**, set **Indent style** to `K&R` or `Allman`.
5. On the **All Settings** tab, under **Advanced**, set **Indent size** to an integer from 2 to 8.

Specify Indent Style Using the Command-Line Interface

1. Create a code configuration object for `'lib'`, `'dll'`, or `'exe'`. For example:

   ```
   cfg = coder.config('lib','ecoder',true); % or dll or exe
   ```
2. Set the `IndentStyle` property to `'K&R'` or `'Allman'`. For example:

   ```
   cfg.IndentStyle = 'Allman';
   ```
3. Set the `IndentSize` property to an integer from 2 to 8. For example:

   ```
   cfg.IndentSize = 4;
   ```

See Also

coder.EmbeddedCodeConfig

More About

- “Configure Build Settings” (MATLAB Coder)

Generate Custom File and Function Banners for C/C++ Code

When you generate C and C++ code from MATLAB code, you can use a code generation template (CGT) file to specify custom:

- File banners
- Function Banners
- File trailers
- Comments before code sections

This example shows how you can create your own CGT file and customize it to generate your own file and function banners.

1. Create a local copy of the default CGT file for MATLAB Coder and rename it. The default CGT file is matlabcoder_default_template.cgt in the *`matlabroot`*`/toolbox/coder/matlabcoder/templates/` folder.
2. Store the copy in a folder that is outside of the MATLAB folder structure, but on the MATLAB path. If necessary, add the folder to the MATLAB path. If you intend to use the CGT file with a custom target, locate the CGT file in a folder under your target root folder. If the file is not on the MATLAB path, specify a full path to the file when adding the file to your configuration.
3. View the default template and generated output. For example, here is the default File Banner section:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Custom File Banner section (optional)
%% Customize File banners by using either custom tokens or the following
%% predefined tokens:
%% %<FileName>, %<MATLABCoderVersion>, %<EmbeddedCoderVersion>
%% %<SourceGeneratedOn>, %<HardwareSelection>, %<OutputType>
%%
%% You can also use "custom tokens" in all of the sections below. See the
%% documentation center for more details.
%%
<FileBanner style="classic">
File: %<FileName>

MATLAB Coder version           : %<MATLABCoderVersion>
C/C++ source code generated on : %<SourceGeneratedOn>
</FileBanner>
```

When you generate code using this default, the file banner looks similar to this file banner:

```
/*
 * File: coderand.c
```

```
 *
 * MATLAB Coder version            : 2.7
 * C/C++ source code generated on  : 06-Apr-2014 14:34:15
 */
```

4 Edit your local copy of the CGT file. You can change the default values and add your own custom tokens. For example, here is the File Banner section with the style changed to `box` and a custom token `myCustomToken`:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Custom File Banner section (optional)
%% Customize File banners by using either custom tokens or the following
%% predefined tokens:
%% %<FileName>, %<MATLABCoderVersion>, %<EmbeddedCoderVersion>
%% %<SourceGeneratedOn>, %<HardwareSelection>, %<OutputType>
%%
%% You can also use "custom tokens" in all of the sections below. See the
%% documentation center for more details.
%%
<FileBanner style="box">
File: %<FileName>

My custom token                : %<myCustomToken>

MATLAB Coder version           : %<MATLABCoderVersion>
C/C++ source code generated on : %<SourceGeneratedOn>
</FileBanner>
```

For more information, see “Code Generation Template Files for MATLAB Code” on page 75-9.

5 Create a configuration object for generation of a C static library for an embedded target.

```
% Create configuration object for an embedded target
cfgObj = coder.config('lib','ecoder',true);
```

6 Create a `MATLABCodeTemplate` object from your CGT file and add it to the configuration object.

```
% Specify the custom CGT file
CGTFile = 'myCGTFile.cgt';
% Use custom template
cfgObj.CodeTemplate = coder.MATLABCodeTemplate(CGTFile);
```

7 Assign values for custom tokens that you added to the template. For example, assign the value `'myValue'` to the `myCustomToken` token that you added in a previous step.

```
cfgObj.CodeTemplate.setTokenValue('myCustomToken','myValue');
```

8 Generate code using the configuration object that you created.

```
codegen -config cfgObj coderand
```

9 View the changes to the generated file banner. For example, here is the file banner for `coderand.c` using the customized CGT file:

```
/*************************************************************************/
/* File: coderand.c                                                      */
/*                                                                       */
/* My custom token               : myValue                               */
/*                                                                       */
/* MATLAB Coder version          : 2.7                                   */
/* C/C++ source code generated on : 06-Apr-2014 14:42:55                 */
/*************************************************************************/
```

Changes to a CGT file do not affect the generated code unless you create a `MATLABCodeTemplate` object from the modified CGT file, and then add it to the configuration object. If you modify the CGT File, `myCGTFile.cgt`, used in the previous example, you must repeat these steps:

1 Create a `MATLABCodeTemplate` object from `myCGTFile.cgt` and add it to the configuration object.

```
CGTFile = 'myCGTFile.cgt';
cfgObj.CodeTemplate = coder.MATLABCodeTemplate(CGTFile);
```

2 Assign the value `'myValue'` to the `myCustomToken` token.

```
cfgObj.CodeTemplate.setTokenValue('myCustomToken','myValue');
```

3 Generate code.

```
codegen -config cfgObj coderand
```

See Also

coder.MATLABCodeTemplate

More About

- “Code Generation Template Files for MATLAB Code” on page 75-9

Code Generation Template Files for MATLAB Code

In this section...

A code generation template (CGT) file defines the sections in generated code that you can customize using comments and tokens. Using a code generation template (CGT) file for the generation of C and C++ code from MATLAB code, you can specify custom file banners and function banners for generated code. File banners are comment sections in the header and trailer sections of a generated file. Function banners are comment sections for each function in the generated code. You can also customize comments before code sections. Use these banners to:

- Add a company copyright statement.
- Specify a special version symbol for your configuration management system.
- Remove time stamps.
- Add other custom information to your generated files.

For information on creating, customizing, and using a CGT file, see "Generate Custom File and Function Banners for C/C++ Code" on page 75-6.

Default CGT File

You can base your custom template on the default CGT file, matlabcoder_default_template.cgt, in the *`matlabroot`*`/toolbox/coder/matlabcoder/templates/` folder.

Note: If you choose not to customize banners for your generated code, the default template is used for code generation.

CGT File Structure

A CGT file consists of 13 optional sections.

File Banner Section

Contains comments and tokens for use in generating a custom file banner.

Function Banner Section

Contains comments and tokens for use in generating a custom function banner.

Shared Utility Function Banner

Contains comments and tokens for use in generating custom banners for shared utility functions.

File Trailer Section

Contains comments for use in generating a custom trailer banner.

Include Files Banner

Contains comments for use in generating a custom banner for the include files section.

Type Definitions

Contains comments for use in generating a custom banner for the type definitions section.

Named Constants

Contains comments for use in generating a custom banner for the named constants section.

Variable Declarations

Contains comments for use in generating a custom banner for the variable declarations section.

Variable Definitions

Contains comments for use in generating a custom banner for the variable definitions section.

Function Declarations

Contains comments for use in generating a custom banner for the function declarations section.

Function Definitions

Contains comments for use in generating a custom banner for the function definitions section.

Custom Source Code

Contains comments for use in generating a custom banner for the custom source code section.

Custom Header Code

Contains comments for use in generating a custom banner for the custom header code section.

Components of the CGT File Sections

Each CGT file section is defined by open and close tags.

CGT File Section	Open Tag	Close Tag
"File Banner" on page 75-14	`<FileBanner>`	`</FileBanner>`
"Function Banner Section" on page 75-10	`<FunctionBanner>`	`</FunctionBanner>`
"Shared Utility Function Banner" on page 75-10	`<SharedUtilityBanner>`	`</SharedUtilityBanner>`
"File Trailer Section" on page 75-10	`<FileTrailer>`	`</FileTrailer>`
"Include Files Banner" on page 75-10	`<IncludeFilesBanner>`	`</IncludeFilesBanner>`
"Type Definitions" on page 75-10	`<TypeDefinitionsBanner>`	`</TypeDefinitionsBanner>`
"Named Constants" on page 75-10	`<NamedConstantsBanner>`	`</NamedConstantsBanner>`
"Variable Declarations" on page 75-10	`<VariableDeclarationsBanner>`	`</VariableDeclarationsBanner>`

CGT File Section	Open Tag	Close Tag
"Variable Definitions" on page 75-10	`<VariableDefinitionsBanner>`	`</VariableDefinitionsBanner>`
"Function Declarations" on page 75-10	`<FunctionDeclarationsBanner>`	`</FunctionDeclarationsBanner>`
"Function Definitions" on page 75-11	`<FunctionDefinitionsBanner>`	`</FunctionDefinitionsBanner>`
"Custom Source Code" on page 75-11	`<CustomSourceCodeBanner>`	`</CustomSourceCodeBanner>`
"Custom Header Code" on page 75-11	`<CustomHeaderCodeBanner>`	`</CustomHeaderCodeBanner>`

You can customize your banners by including tokens and comments between the open and close tags for each section. Tokens are replaced with values in the generated code. The following rules apply to tokens in your CGT file:

- You can have only one token per line.
- Token values must not contain a '\t' for formatting.

Note: In the contents of your banner, C comment indicators, '/*' or '*/', can introduce an error in the generated code.

An open tag includes tag attributes. Enclose the value of the attribute in double quotes. The attributes available for an open tag are:

- `width`: specifies the width of the file or function banner comments in the generated code. The default value is 80.
- `style`: specifies the boundary for the file or function banner comments in the generated code.

The open tag syntax is:

```
<OpenTag style = "style_value" width = "num_width">
```

There are five options for the banner style. The `CommentStyle` and `TargetLang` configuration object properties determine the use of C or C++ comment style. The built-in style options for the `style` attribute are:

- `classic`

  Using C style comments

```
/* single line comments */

/*
 * multiple line comments
 * second line
 */
```

  Using C++ style comments

```
// single line comments

//
// multiple line comments
// second line
//
```

- `box`

  Using C style comments

```
/*********************************************************/
/* banner contents                                       */
/*********************************************************/
```

  Using C++ style comments

```
///////////////////////////////////////////////////////////
// banner contents                                       //
///////////////////////////////////////////////////////////
```

- `open_box`

  Using C style comments

```
/*********************************************************
 * banner contents
 *********************************************************/
```

  Using C++ style comments

```
///////////////////////////////////////////////////////////
// banner contents
```

```
////////////////////////////////////////////////////////////
```

- `doxygen`

  Using C style comments

  ```
  /** single line comments */

  /**
   * multiple line comments
   * second line
   */
  ```

  Using C++ style comments

  ```
  ///single line comments

  ///
  /// multiple line comments
  ///second line
  ///
  ```

- `doxygen_qt`

  Using C style comments

  ```
  /*! single line comments */

  /*!
   * multiple line comments
   * second line
   */
  ```

  Using C++ style comments

  ```
  //!single line comments

  //!
  //! multiple line comments
  //!second line
  //!
  ```

File Banner

This section contains comments and tokens for use in generating a custom file banner that precedes the generated C and C++ code. If you omit the file banner section from the

CGT file, the code generator does not generate a file banner in the generated code. The file banner section provided in the default CGT file is:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Custom File Banner section (optional)
%% Customize File banners by using either custom tokens or the following
%% predefined tokens:
%% %<FileName>, %<MATLABCoderVersion>, %<EmbeddedCoderVersion>
%% %<SourceGeneratedOn>, %<HardwareSelection>, %<OutputType>
%%
%% You can also use "custom tokens" in all of the sections below. See the
%% documentation center for more details.
%%
<FileBanner style="classic">
File: %<FileName>

MATLAB Coder version           : %<MATLABCoderVersion>
C/C++ source code generated on : %<SourceGeneratedOn>
</FileBanner>
```

Summary of Tokens for File Banner Generation

`FileName`	Name of the generated file (for example, `"kalman.c"`)
`SourceGeneratedOn`	Time stamp of generated file
`MATLABCoderVersion`	Version of MATLAB Coder
`EmbeddedCoderVersion`	Version of Embedded Coder
`HardwareSelection`	Selected target
`OutputType`	Type of output (for example, lib, exe, or dll)

Function Banner

This section contains comments and tokens for use in generating a custom function banner that precedes a generated C or C++ function. If you omit the function banner section from the CGT file, the code generator does not generate function banners. The function banner section provided in the default CGT file is:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Custom function banner section (optional)
%% Customize function banners by using the following predefined tokens:
%% %<FunctionName>, %<FunctionDescription>
%% %<Arguments>, %<ReturnType>
%%
<FunctionBanner style="classic">
%<FunctionDescription>
Arguments    : %<Arguments>
Return Type  : %<ReturnType>
%</FunctionBanner>
```

Summary of Tokens for Function Banner Generation

`FunctionName`	Name of function
`FunctionDescription`	Short abstract about the function
`Arguments`	List of function arguments
`ReturnType`	Return type of function

Shared Utility Banner

This section contains comments and tokens for use in generating a custom shared utility function banner that precedes a generated C or C++ shared utility function. If you omit the shared utility function banner section from the CGT file, the code generator does not generate shared utility function banners. The shared utility function banner section provided in the default CGT file is:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Custom Shared Utility Function Banner section (optional)
%% Customize shared utility function banners by using the following
%% predefined tokens:
%% %<FunctionName>, %<FunctionDescription>
%% %<Arguments>, %<ReturnType>
%%
<SharedUtilityBanner style="classic">
Arguments    : %<Arguments>
Return Type  : %<ReturnType>
</SharedUtilityBanner>
```

Summary of Tokens for Shared Utility Function Banner Generation

`FunctionName`	Name of function
`FunctionDescription`	Short abstract about the function
`Arguments`	List of function arguments
`ReturnType`	Return type of function

File Trailer

The file trailer section contains comments for generating a custom file trailer that follows the generated C or C++ code. If you omit the file trailer section from the CGT file, the code generator does not generate a file trailer. The file trailer section provided in the default CGT file is:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Custom file trailer section (optional)
```

```
%% You can use any of the predefined tokens used for File Banner
%%
<FileTrailer style="classic">
File trailer for %<FileName>

[EOF]
</FileTrailer>
```

Tokens for the file banner are available for the file trailer. See Summary of Tokens for File Banner Generation.

Include Files Banner

The include files banner section contains comments for generating a custom banner that precedes the include files section in the generated code. If you omit the include files banner section from the CGT file, the code generator does not generate a banner for this section. The include files banner section provided in the default CGT file is:

```
<IncludeFilesBanner  style="classic">
Include Files
</IncludeFilesBanner>
```

Type Definitions Banner

The type definitions banner section contains comments for generating a custom banner that precedes the type definitions section in the generated code. If you omit the type definitions banner section from the CGT file, the code generator does not generate a banner for this section. The type definitions banner section provided in the default CGT file is:

```
<TypeDefinitionsBanner  style="classic">
Type Definitions
</TypeDefinitionsBanner>
```

Named Constants Banner

The named constants banner section contains comments for generating a custom banner that precedes the named constants section in the generated code. If you omit the named constants banner section from the CGT file, the code generator does not generate a banner for this section. The named constants banner section provided in the default CGT file is:

```
<NamedConstantsBanner  style="classic">
Named Constants
</NamedConstantsBanner>
```

Variable Declarations

The variable declarations banner section contains comments for generating a custom banner that precedes the variable declarations section in the generated code. If you omit the variable declarations banner section from the CGT file, the code generator does not generate a banner for this section. The variable declarations banner section provided in the default CGT file is:

```
<VariableDeclarationsBanner  style="classic">
Variable Declarations
</VariableDeclarationsBanner>
```

Variable Definitions

The variable definitions banner section contains comments for generating a custom banner that precedes the variable definitions section in the generated code. If you omit the variable definitions banner section from the CGT file, the code generator does not generate a banner for this section. The variable definitions banner section provided in the default CGT file is:

```
<VariableDefinitionsBanner  style="classic">
Variable Definitions
</VariableDefinitionsBanner>
```

Function Declarations

The function declarations banner section contains comments for generating a custom banner that precedes the function declarations section in the generated code. If you omit the function declarations banner section from the CGT file, the code generator does not generate a banner for this section. The function declarations banner section provided in the default CGT file is:

```
<functionDeclarationsBanner  style="classic">
Function Declarations
</FunctionDeclarationsBanner>
```

Function Definitions

The function definitions banner section contains comments for generating a custom banner that precedes the function definitions section in the generated code. If you omit the function definitions banner section from the CGT file, the code generator does not generate a banner for this section. The function definitions banner section provided in the default CGT file is:

```
<FunctionDefinitionsBanner  style="classic">
Function Definitions
</FunctionDefinitionsBanner>
```

Custom Source Code

The custom source code banner section contains comments for generating a custom banner that precedes the custom source code section in the generated code. If you omit the custom source code banner section from the CGT file, the code generator does not generate a banner for this section. The custom source code banner section provided in the default CGT file is:

```
<CustomSourceCodeBanner  style="classic">
Custom Source Code
</CustomSourceCodeBanner>
```

Custom Header Code

The custom header code banner section contains comments for generating a custom banner that precedes the custom header code section in the generated code. If you omit the custom header code banner section from the CGT file, the code generator does not generate a banner for this section. The custom header code banner section provided in the default CGT file is:

```
<CustomHeaderCodeBanner  style="classic">
Custom Header Code
</CustomHeaderCodeBanner>
```

See Also

coder.MATLABCodeTemplate

More About

- “Generate Custom File and Function Banners for C/C++ Code” on page 75-6

Customize Generated Identifiers

In this section...

If you have Embedded Coder, you can customize the identifiers in C/C++ code generated from MATLAB code. For each kind of identifier that you want to customize, set the appropriate identifier format parameter to a macro that specifies the format of the generated identifiers. The macro can include:

- Valid C or C++ language identifiers (a-z, A-Z, _, 0–9).
- The tokens listed in the following table. `$M` is required.

Token	Description
`$M`	Code generator inserts name mangling text to avoid naming collisions. Required.
`$N`	Code generator inserts the name of the object (global variable, global type, local function, local temporary variable, or constant macro) for which the identifier is generated. Improves readability of generated code.
`$R`	Code generator inserts the root project name into identifier, replacing unsupported characters with the underscore (_) character.

Customize Identifiers by Using the MATLAB Coder App

1. On the **Generate Code** page, to open the **Generate** dialog box, click the **Generate** arrow  .
2. Set **Build type** to one of the following:
   - `Source Code`
   - `Static Library`

- `Dynamic Library`
- `Executable`

3 Click **More Settings**.

4 On the **Code Appearance** tab, under **Identifier Format**, for each kind of identifier that you want to customize, enter the macro.

Parameter	Default Macro
Global variables	`$M$N`
Global types	`$M$N`
Field name of global types	`$M$N`
Local functions	`$M$N`
Local temporary variables	`$M$N`
Constant macros	`$M$N`
EMX Array Types	`emxArray_$M$N`
EMX Array Utility Functions	`emx$M$N`

For example, suppose that **Global variables** has the value `glob_$M$N`. For a global variable named `g`, when name mangling is not required, the generated identifier is `glob_g`. If name mangling is required, the generated identifier includes the name mangling text.

Customize Generated Identifiers by Using the Command-Line Interface

1 Create a code configuration object for a library or executable program. For example:

```
cfg = coder.config('lib','ecoder',true);
```

2 For each kind of identifier that you want to customize, specify the macro as a character vector.

Parameter	Description	Default Macro
`CustomSymbolStrGlobalVar`	Global variables	`'$M$N'`
`CustomSymbolStrType`	Global types	`'$M$N'`
`CustomSymbolStrField`	Field name of global types	`'$M$N'`
`CustomSymbolStrFcn`	Local functions	`'$M$N'`

Parameter	Description	Default Macro
`CustomSymbolStrTmpVar`	Local temporary variables	`'$M$N'`
`CustomSymbolStrMacro`	Constant macros	`'$M$N'`
`CustomSymbolStrEMXArray`	EMX Array Types	`'emxArray_$M$N'`
`CustomSymbolStrEMXArrayFcn`	EMX Array Utility Functions	`'emx$M$N'`

For example:

```
cfg.CustomSymbolStrGlobalVar = 'glob_$M$N';
```

For a global variable named `g`, when name mangling is not required, the generated identifier is `glob_g`. If name mangling is required, the generated identifier includes the name mangling text.

See Also

coder.EmbeddedCodeConfig

More About

- "Configure Build Settings" (MATLAB Coder)

Control Signed Left Shifts in Generated Code

In this section...

If you have an Embedded Coder license, you can control whether MATLAB Coder replaces multiplications by powers of two with signed left bitwise shifts. Some coding standards, such as MISRA, do not allow bitwise operations on signed integers.

By default, MATLAB Coder replaces multiplication by powers of two with signed left shifts. Here is an example of generated C code that uses a signed left shift for multiplication by eight.

```
i <<= 3;
```

To increase the likelihood of generating MISRA C:2012 compliant code, disable the replacement of multiplication by powers of two with signed left shifts. Here is an example of generated C code that does not use a signed left shift for multiplication by eight:

```
i = i * 8;
```

Control Signed Left Shifts Using the MATLAB Coder App

1. On the **Generate Code** page, to open the **Generate** dialog box, click the **Generate** arrow  .
2. Set **Build type** to one of the following:
   - `Source Code`
   - `Static Library (.lib)`
   - `Dynamic Library (.dll)`
   - `Executable (.exe)`
3. Click **More Settings**.
4. On the **Code Appearance** tab, select or clear the **Use signed shift left for fixed-point operations and multiplication by powers of 2** check box.

Control Signed Left Shifts Using the Command-Line Interface

1. Create a code configuration object for `'lib'`, `'dll'`, or `'exe'`. For example:

```
cfg = coder.config('lib','ecoder',true); % or dll or exe
```

2 Set the `EnableSignedLeftShifts` property to `true` or `false`. For example:

```
cfg.EnableSignedLeftShifts = false;
```

See Also

coder.EmbeddedCodeConfig

More About

- "Configure Build Settings" (MATLAB Coder)

Control Data Type Casts in Generated Code

In this section...

If you have an Embedded Coder license, you can control data type casts in C/C++ code generated from MATLAB code. You can specify one of the following casting modes.

Casting Mode	Description

Nominal	Nominal casting mode is the default casting mode. Generated C/C++ code uses the default C compiler data type casting. When you do not have special data type information requirements, choose this option. Here is an example of code generated using nominal casting mode:
	`short addone(short x)`
	`{`
	`  int i0;`
	`  i0 = x + 1;`
	`  if (i0 > 32767) {`
	`  i0 = 32767;`
	`  }`
	`  return (short)i0;`
	`}`

Standards Compliant	Generated C/C++ code has data type casts that conform to MISRA standards. The MISRA data type casting eliminates common MISRA standard violations, including address arithmetic and assignment. It reduces 10.1, 10.2, 10.3, and 10.4 violations. Here is an example of code generated using standards-compliant casting mode: `short addone(short x)`

Casting Mode	Description

	{
	  int i0;
	  i0 = (int)x + (int)1;
	  if (i0 > (int)32767) {
	  i0 = (int)32767;
	  }
	
	  return (short)i0;
	}

Explicit	Generated C/C++ code has explicit data type casts. Explicit data type casts provide information about the amount of memory that the variable uses and the level of precision for calculations using the variable. Here is an example of code generated using explicit casting mode:
	
	short addone(short x)
	{
	  int i0;
	  i0 = (int)x + 1;
	  if (i0 > 32767) {
	  i0 = 32767;
	  }
	
	  return (short)i0;
	}

Specify Casting Mode Using the MATLAB Coder App

1. On the **Generate Code** page, to open the **Generate** dialog box, click the **Generate** arrow  .
2. Set **Build type** to one of the following:
   - `Source Code`
   - `Static Library (.lib)`
   - `Dynamic Library (.dll)`
   - `Executable (.exe)`
3. Click **More Settings**.

4 On the **All Settings** tab, under **Advanced**, set **Casting mode** to one of the following values:

- `Nominal`
- `Standards Compliant`
- `Explicit`

Specify Casting Mode Using the Command-Line Interface

1 Create a code configuration object for `'lib'`, `'dll'`, or `'exe'`. For example:

```
cfg = coder.config('lib','ecoder',true); % or dll or exe
```

2 Set the `CastingMode` property to one of the following values:

- `'Nominal'`
- `'Standards'`
- `'Explicit'`

For example:

```
cfg.IndentStyle = 'Standards';
```

See Also

coder.EmbeddedCodeConfig

More About

- "Configure Build Settings" (MATLAB Coder)

Simplify Multiply Operations for Array Indexing in Loops

If you use Embedded Coder to generate C/C++ code from MATLAB code, you can enable an optimization that simplifies array indexing in loops in the generated code. When possible, for array indices in loops, this optimization replaces multiply operations with add operations. Multiply operations can be expensive. This optimization, referred to as strength reduction, is useful when the C/C++ compiler on the target platform does not optimize the array indexing.

Here is code generated without the optimization:

```
for (i = 0; i < 10; i++) {
    z[5 * (1 + i) - 1] = x[5 * (1 + i)];
  }
```

Here is code generated with the optimization:

```
for (b_i = 0; b_i < 10; b_i++) {
    z[i + 4] = x[i + 5];
    i += 5;
  }
```

By default, the strength reduction optimization is disabled. To enable it:

- At the command line, set the configuration object parameter `EnableStrengthReduction` to `true`.
- In the MATLAB Coder app, project build settings, on the **All Settings** tab, set **Simplify array indexing** to `Yes`.

Even when the optimization replaces the multiply operations in the generated code, it is possible that the C/C++ compiler can generate multiply instructions.

More About

- "Optimization Strategies" (MATLAB Coder)
- "Configure Build Settings" (MATLAB Coder)

Code Replacement for MATLAB Code

What Is Code Replacement?

Code replacement is a technique to change the code that the code generator produces for functions and operators to meet application code requirements. For example, you can replace generated code to meet requirements such as:

- Optimization for a specific run-time environment, including, but not limited to, specific target hardware.
- Integration with existing application code.
- Compliance with a standard, such as AUTOSAR.
- Modification of code behavior, such as enabling or disabling nonfinite or inline support.
- Application- or project-specific code requirements, such as:
  - Elimination of `math.h`.
  - Elimination of system header files.
  - Elimination of calls to `memcpy` or `memset`.
  - Use of BLAS.
  - Use of a specific BLAS.

To apply this technique, configure the code generator to apply a code replacement library (CRL) during code generation. By default, the code generator does not apply a code replacement library. You can choose from the following libraries that MathWorks provides:

- GNU C99 extensions—GNU[16] gcc math library, which provides C99 extensions as defined by compiler option `-std=gnu99`.
- AUTOSAR 4.0—Produces code that more closely aligns with the AUTOSAR standard. Requires an Embedded Coder license.
- Intel IPP for x86-64 (Windows)—Generates calls to the Intel Performance Primitives (IPP) library for the x86-64 Windows platform.
- Intel IPP/SSE for x86-64 (Windows)—Generates calls to the IPP and Streaming SIMD Extensions (SSE) libraries for the x86-64 Windows platform.
- Intel IPP for x86-64 (Windows using MinGW compiler)—Generates calls to the IPP library for the x86-64 Windows platform and MinGW compiler.

16. GNU is a registered trademark of the Free Software Foundation.

- Intel IPP/SSE for x86-64 (Windows using MinGW compiler)—Generates calls to the IPP and SSE libraries for the x86-64 Windows platform and MinGW compiler.
- Intel IPP for x86/Pentium (Windows)—Generates calls to the IPP library for the x86/Pentium Windows platform.
- Intel IPP/SSE for x86/Pentium (Windows)—Generates calls to the Intel Performance IPP and SSE libraries for the x86/Pentium Windows platform.
- Intel IPP for x86-64 (Linux)—Generates calls to the IPP library for the x86-64 Linux platform.
- Intel IPP/SSE with GNU99 extensions for x86-64 (Linux)—Generates calls to the GNU libraries for IPP and SSE, with GNU C99 extensions, for the x86-64 Linux platform.

Libraries that include GNU99 extensions are intended for use with the GCC compiler. If you use one of those libraries with another compiler, generated code might not compile.

Depending on the product licenses that you have, other libraries might be available . If you have an Embedded Coder license, you can view and choose from other libraries and you can create custom code replacement libraries.

Code Replacement Libraries

A *code replacement library* consists of one or more code replacement tables that specify application-specific implementations of functions and operators. For example, a library for a specific embedded processor specifies function and operator replacements that optimize generated code for that processor.

A *code replacement table* contains one or more *code replacement entries*, with each entry representing a potential replacement for a function or operator. Each entry maps a *conceptual representation* of a function or operator to an *implementation representation* and priority.

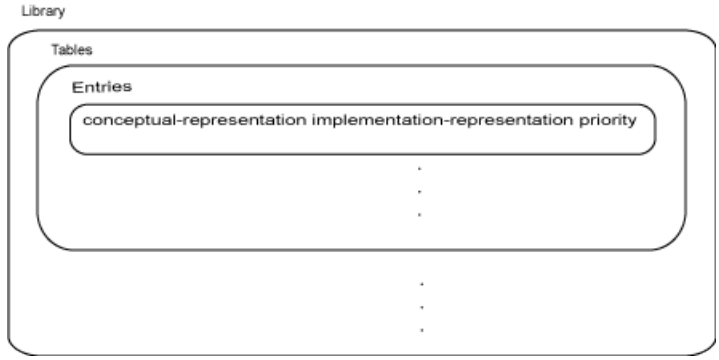


Table Entry Component	Description
Conceptual representation	Identifies the table entry and contains match criteria for the code generator. Consists of: • Function name or a key. The function name identifies most functions. For operators and some functions, a series of characters, called a key identifies a function or operator. For example, function name `'cos'` and operator key `'RTW_OP_ADD'`. • Conceptual arguments that observe code generator naming (`'y1'`, `'u1'`, `'u2'`, ...), with corresponding I/O types (output or input) and data types. • Other attributes, such as an algorithm, fixed-point saturation, and rounding modes, which identify matching criteria for the function or operator.
Implementation representation	Specifies replacement code. Consists of: • Function name. For example, `'cos_dbl'` or `'u8_add_u8_u8'`. • Implementation arguments, with corresponding I/O types (output or input) and data types. • Parameters that provide additional implementation details, such as header and source file names and paths of build resources.

Table Entry Component	Description
Priority	Defines the entry priority relative to other entries in the table. The value can range from 0 to 100, with 0 being the highest priority. If multiple entries have the same priority, the code generator uses the first match with that priority.

When the code generator looks for a match in a code replacement library, it creates and populates a *call site object* with the function or operator conceptual representation. If a match exists, the code generator uses the matched code replacement entry populated with the implementation representation and uses it to generate code.

The code generator searches the tables in a code replacement library for a match in the order that the tables appear in the library. If the code generator finds multiple matches within a table, the priority determines the match. The code generator uses a higher-priority entry over a similar entry with a lower priority.

Code Replacement Terminology

Term	Definition
Cache hit	A code replacement entry for a function or operator, defined in the specified code replacement library, for which the code generator finds a match.
Cache miss	A conceptual representation of a function or operator for which the code generator does not find a match.
Call site object	Conceptual representation of a function or operator that the code generator uses when it encounters a call site for a function or operator. The code generator uses the object to query the code replacement library for a conceptual representation match. If a match exists, the code generator returns a code replacement object, fully populated with the conceptual representation, implementation representation, and priority, and uses that object to generate replacement code.
Code replacement library	One or more code replacement tables that specify application-specific implementations of functions

Term	Definition
	and operators. When configured to use a code replacement library, the code generator uses criteria defined in the library to search for matches. If a match is found, the code generator replaces code that it generates by default with application-specific code defined in the library.
Code replacement table	One or more code replacement table entries. Provides a way to group related or shared entries for use in different libraries.
Code replacement entry	Represents a potential replacement for a function or operator. Maps a conceptual representation of a function or operator to an implementation representation and priority.
Conceptual argument	Represents an input or output argument for a function or operator being replaced. Conceptual arguments observe naming conventions (`'y1'`, `'u1'`, `'u2'`, ...) and data types familiar to the code generator.
Conceptual representation	Represents match criteria that the code generator uses to qualify functions and operators for replacement. Consists of: • Function or operator name or key • Conceptual arguments with type, dimension, and complexity specification for inputs and output • Attributes, such as an algorithm and fixed-point saturation and rounding modes
Implementation argument	Represents an input or output argument for a C or C++ replacement function. Implementation arguments observe C/C++ name and data type specifications.

Term	Definition
Implementation representation	Specifies C or C++ replacement function prototype. Consists of: • Function name (for example, `'cos_dbl'` or `'u8_add_u8_u8'`) • Implementation arguments specifying type, type qualifiers, and complexity for the function inputs and output • Parameters that provide build information, such as header and source file names and paths of build resources and compile and link flags
Key	Identifies a function or operator that is being replaced. A function name or key appears in the conceptual representation of a code replacement entry. The key `RTW_OP_ADD` identifies the addition operator.
Priority	Defines the match priority for a code replacement entry relative to other entries, which have the same name and conceptual argument list, within a code replacement library. The priority can range from 0 to 100, with 0 being the highest priority. The default is 100. If a library provides two implementations for a function or operator, the implementation with the higher priority shadows the one with the lower priority.

Code Replacement Limitations

Code replacement verification — It is possible that code replacement behaves differently than you expect. For example, data types that you observe in code generator input might not match what the code generator uses as intermediate data types during an operation. Verify code replacements by examining generated code.

Code replacement for matrices — Code replacement libraries do not support Dynamic and Symbolic sized matrices.

Related Examples

- "Replace Code Generated from MATLAB Code" on page 76-11
- "Choose a Code Replacement Library" on page 76-9

Choose a Code Replacement Library

In this section...

About Choosing a Code Replacement Library

By default, the code generator does not use a code replacement library.

If you are considering using a code replacement library:

1. Explore available libraries. Identify one that best meets your application needs.
   - Consider the lists of application code replacement requirements and libraries that MathWorks provides in "What Is Code Replacement?" on page 76-2.
   - See "Explore Available Code Replacement Libraries" on page 76-9.
2. Explore the contents of the library. See "Explore Code Replacement Library Contents" on page 37-9.

If you do not find a suitable library and you have an Embedded Coder license, you can create a custom code replacement library.

Explore Available Code Replacement Libraries

You can select the code replacement library to use for code generation in a project, on the **Custom Code** tab, by setting the **Code replacement library** parameter. Alternatively, in a code configuration object, set the `CodeReplacementLibrary` parameter.

Explore Code Replacement Library Contents

Use the Code Replacement Viewer to explore the content of a code replacement library.

1. At the command prompt, type `crviewer`.

```
>> crviewer
```

The viewer opens. To view the content of a specific library, specify the name of the library as an argument in single quotes. For example:

```
>> crviewer('GNU C99 extensions')
```

2. In the left pane, select the name of a library. The viewer displays information about the library in the right pane.
3. In the left pane, expand the library, explore the list of tables it contains, and select a table from the list. In the middle pane, the viewer displays the function and operator entries that are in that table, along with abbreviated information for each entry.
4. In the middle pane, select a function or operator. The viewer displays information from the table entry in the right pane.

   If you select an operator entry that specifies net slope fixed-point parameters (instantiated from entry class `RTW.TflCOperationEntryGenerator` or `RTW.TflCOperationEntryGenerator_NetSlope`), the viewer displays an additional tab that shows fixed-point settings.

See Code Replacement Viewer for details on what the viewer displays.

Related Examples

- “What Is Code Replacement?” on page 76-2
- “Replace Code Generated from MATLAB Code” on page 76-11

Replace Code Generated from MATLAB Code

This example shows how to replace generated code using a code replacement library. Code replacement is a technique for changing the code that the code generator produces for functions and operators to meet application code requirements.

Prepare for Code Replacement

1. Make sure that you have installed required software. Required software is:

   - MATLAB
   - MATLAB Coder
   - C compiler

   Some code replacement libraries available in your development environment require Embedded Coder.

   For instructions on installing MathWorks products, see the MATLAB installation documentation. If you have installed MATLAB and want to see which other MathWorks products are installed, in the MATLAB Command Window, enter `ver`.
2. Identify an existing MATLAB function or create a new MATLAB function for which you want the code generator to replace code.

Choose a Code Replacement Library

If you are not sure which library to use, explore available libraries.

Configure Code Generator To Use Code Replacement Library

1. Configure the code generator to apply a code replacement library during code generation for the MATLAB function. Do one of the following:

   - In a project, on the **Custom Code** tab, set the **Code replacement library** parameter.
   - In a code configuration object, set the `CodeReplacementLibrary` parameter.
2. Configure the code generator to produce only code. Before you build an executable, verify your code replacements. Do one of the following:

   - In a project, in the **Generate** dialog box, select the **Generate code only** check box.

- In a code configuration object, set the `GenCodeOnly` parameter.

Include Code Replacement Information In Code Generation Report

If you have an Embedded Coder license, you can configure the code generator to include a code replacement section in the code generation report. The additional information helps you verify code replacements.

1. Configure the code generator to generate a report.
   - In a project, on the **Debugging** tab, set the **Always create a code generation report** parameter.
   - In a code configuration object, set the `GenerateReport` parameter.
2. Include the code replacement section in the report.
   - In a project, on the **Debugging** tab, select the **Code replacements** check box.
   - In a code configuration object, set the `GenerateCodeReplacementReport` parameter.

Generate Replacement Code

Generate C/C++ code from the MATLAB code. If you configured the code generator to produce a report, generate a code generation report. For example, in the MATLAB Coder app, on the **Generate Code** page, click **Generate**. Or, at the command prompt, enter:

```
codegen -report myFunction -args {5} -config cfg
```

The code generator produces the code and displays the report.

Verify Code Replacements

Verify code replacements by examining the generated code. Code replacement can sometimes behave differently than you expect. For example, data types that you observe in the code generator input might not match what the code generator uses as intermediate data types during an operation.

Related Examples

- “What Is Code Replacement?” on page 76-2
- “Choose a Code Replacement Library” on page 76-9
- “Configure Build Settings” (MATLAB Coder)

77

Storage Classes for Code Generation from MATLAB Code

Storage Classes for Code Generation from MATLAB Code

If you have an Embedded Coder license, you can use storage classes to control the declaration and definition of a global variable in the generated C/C++ code.

In the context of code generation, a *storage class* is a specification that determines the declaration and definition of a variable in the generated code. For code generation, the term storage class is not the same as the C language term *storage class specifier*.

Storage classes help you to integrate generated code with external code. You can make a generated variable visible to external code. You can also make variables declared in the external code visible to the generated code. For code generation from MATLAB code, you can use storage classes with global variables only. The storage class determines:

- The file placement of a global variable declaration and definition.
- Whether the global variable is imported from external code or exported for use by external code.

To assign a storage class to a global variable, in your MATLAB code, use the `coder.storageClass` function. Only when you use an Embedded Coder project or configuration object for generation of C/C++ libraries or executables does the code generator recognize `coder.storageClass` calls.

The syntax for `coder.storageClass` is:

```
coder.storageClass(global_name, storage_class)
```

`global_name` is the name of a global variable, specified as a character vector. `global_name` must be a compile-time constant.

`storage_class` can be one of the following values.

Storage Class	Description
`'ExportedGlobal'`	• Defines the variable in the `Variable Definitions` section of the C file *`entry_point_name`*`.c`. • Declares the variable as an `extern` in the `Variable Declarations` section of the header file *`entry_point_name`*`.h`

Storage Class	Description
	• Initializes the variable in the function *entry_point_name*_initialize.h.
'ExportedDefine'	Declares the variable with a #define directive in the Exported data define section of the header file *entry_point_name*.h.
'ImportedExtern'	Declares the variable as an extern in the Variable Declarations section of the header file *entry_point_name*_data.h. The external code must supply the variable definition.
'ImportedExternPointer'	Declares the variable as an extern pointer in the Variable Declarations section of the header file *entry_point_name*_data.h. The external code must define a valid pointer variable.

Storage classes have these requirements and limitations:

- Assign the storage class to a global variable in a function that declares the global variable. You do not have to assign the storage class in more than one function.
- After you assign a storage class to a global variable, you cannot assign a different storage class to that global variable.
- You cannot assign a storage class to a constant global variable.
- A global variable with an ExportedDefine storage class must be a scalar but not a complex or multi-word scalar. The global variable must only be read and not written to in the code.

If you do not assign a storage class to a global variable, except for the declaration location, the variable behaves like it has an 'ExportedGlobal' storage class. For an 'ExportedGlobal' storage class, the global variable is declared in the file *entry_point_name*.h. When the global variable does not have a storage class, the variable is declared in the file *entry_point_name*_data.h.

See Also

coder.storageClass

Related Examples

- "Control Declarations and Definitions of Global Variables in Code Generated from MATLAB Code" on page 77-5
- "Generate Code for Global Data" (MATLAB Coder)

Control Declarations and Definitions of Global Variables in Code Generated from MATLAB Code

This example uses storage classes to control the declarations and definitions of global variables in C/C++ code generated from MATLAB code. Using storage classes helps you to interface generated code with external code.

This example requires an Embedded Coder license.

Write a function `addglobals` that adds four global variables. Declare the global variables in the function.

```
function y = addglobals

% Define the global variables.

global u;
global v;
global x;
global z;

% Assign the storage classes.

coder.storageClass('u','ExportedGlobal');
coder.storageClass('v','ImportedExtern');
coder.storageClass('x','ImportedExternPointer');
coder.storageClass('z','ExportedDefine');
y = u + v + x + z;
end
```

Create a file `c:\myfiles\myfile.c` that defines and initializes the imported global variables `u` and `v`.

```
#include <stdio.h>

/* Variable definitions for imported variables */
double v = 1.0;
double *x = &v;
```

Create a code configuration object. Configure the code generation parameters to include `myfile.c`. For output type `'lib'`, or if you generate source code only, you can generate code without providing this file. Otherwise, you must provide this file.

```
cfg = coder.config('dll','ecoder', true);
cfg.CustomSource = 'myfile.c';
cfg.CustomInclude = 'c:\myfiles';
```

Generate the code. This example uses the `-globals` argument to specify the types and initial values of the global variables `u`, `v`, `x`, and `z`. Alternatively, you can define global variables in the MATLAB global workspace. For the imported global variables `v` and `x`, the code generator uses the initial values only to determine the type.

```
codegen -config cfg -globals {'u', 1, 'v', 2, 'x', 3, 'z', 4} addglobals -report
```

From the initial values `1`, `2`, `3`, and `4` `codegen` determines that `u`, `v`, `x` and `z` have type `double`. `codegen` defines and declares the exported global variables `u` and `z`. It generates code that initializes `u` to `1.0` and `z` to `4.0`. `codegen` declares the imported global variables `v` and `x`. It does not define these variables or generate code that initializes them. `myfile.c` provides the code that defines and initializes `v` and `x`.

To view the code generated for the global variables, open the report. Click the `View report` link.

View the definition for the exported global `z` in the `Exported data define` section in `addglobals.h`.

```
/* Definition for custom storage class: ExportedDefine */
#define z                              4.0
```

View the definition and declaration for the exported global `u`.

- `u` is defined in the `Variable Definitions` section in `addglobals.c`.

  ```
  /* Variable Definitions */
  /* Definition for custom storage class: ExportedGlobal */
  double u;
  ```

- `u` is declared as `extern` in the `Variable Declarations` section in `addglobals.h`.

  ```
  /* Variable Declarations */
  /* Declaration for custom storage class: ExportedGlobal */
  extern double u;
  ```

- `u` is initialized in `addglobals_initialize.c`.

  ```
  /* Include Files */
  #include "rt_nonfinite.h"
  #include "addglobals.h"
  ```

```
#include "addglobals_initialize.h"

/* Named Constants */
#define b_u                            (1.0)

/* Function Definitions */

/*
 * Arguments    : void
 * Return Type  : void
 */
void addglobals_initialize(void)
{
  rt_InitInfAndNaN(8U);
  u = b_u;
}
```

View the definition and declaration for the imported external global `v` and the imported external global pointer `x`.

`v` and `x` are declared as `extern` in the `Variable Declarations` section in `addglobals_data.h`.

```
/* Variable Declarations */
/* Declaration for custom storage class: ImportedExtern */
extern double v;

/* Declaration for custom storage class: ImportedExternPointer */
extern double *x;
```

See Also

`coder.storageClass`

More About

- “Storage Classes for Code Generation from MATLAB Code” on page 77-2
- “Generate Code for Global Data” (MATLAB Coder)
- “Specify External File Locations” (MATLAB Coder)

Verification of Code Generated from MATLAB Code

Highlight Potential Data Type Issues in a Report

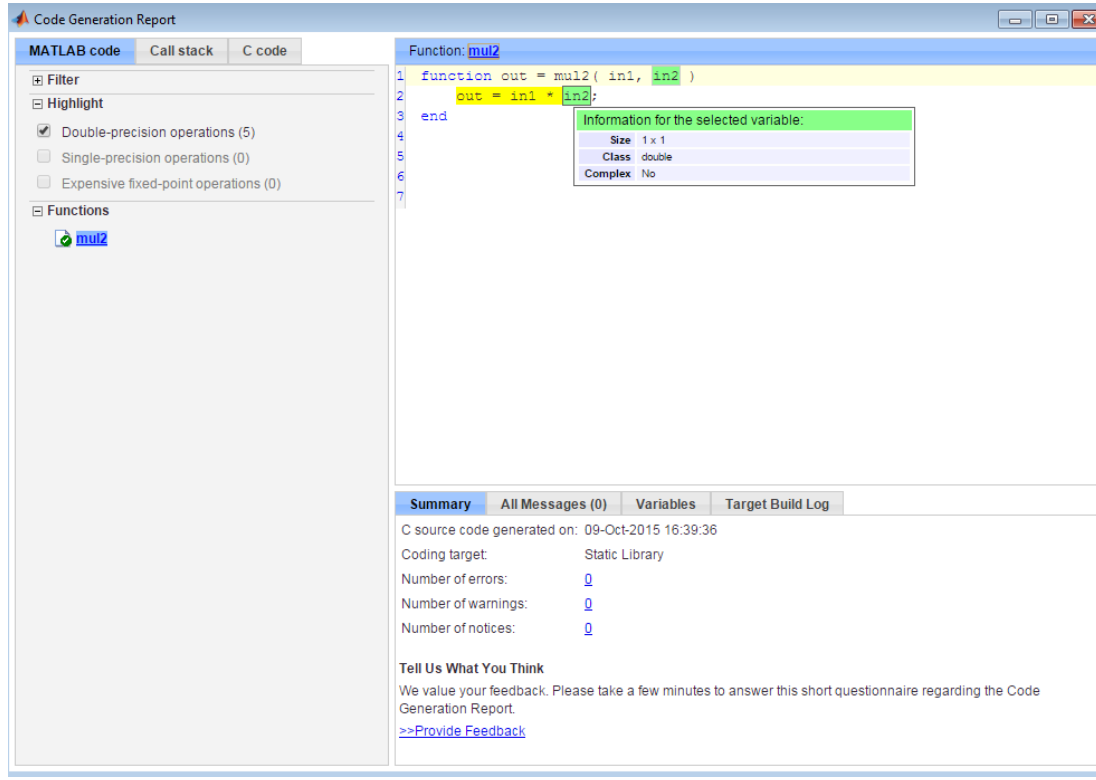
In this section...

If you have an Embedded Coder license, you have the option to highlight potential data types issues in the code generation report for standalone code generated from MATLAB code. If you enable this option, the **Highlight** section on the **MATLAB code** tab lists the number of single-precision and double-precision operations in the generated C/C++ code. If you have a Fixed-Point Designer license, it also lists the number of expensive fixed-point operations.

To highlight the MATLAB code that corresponds to the potential data type issues:

1. Select the check box for the type of operation that you want to highlight.
2. Select the function that you want to highlight.

   The report highlights the operations in the selected function. The following example report highlights MATLAB code that results in double-precision operations in the generated code.



The option to highlight potential data type issues is disabled by default.

Enable Highlight Option Using the MATLAB Coder App

1. On the **Generate Code** page, to open the **Generate** dialog box, click the **Generate** arrow  .
2. Set **Build type** to one of the following:
   - `Source Code`
   - `Static Library (.lib)`
   - `Dynamic Library (.dll)`
   - `Executable (.exe)`

3 Click **More Settings**.

4 On the **Debugging** tab, select the **Always create a code generation report** and **Highlight potential data type issues** check boxes.

Enable Highlight Option Using the Command Line Interface

1 Create an embedded code configuration object for `'lib'`, `'dll'`, or `'exe'`:

```
cfg = coder.config('lib','ecoder',true); % or dll or exe
```

2 Set the `GenerateReport` and `HighlightPotentialDataTypeIssues` configuration object properties to `true`:

```
cfg.GenerateReport = true;
cfg.HighlightPotentialDataTypeIssues = true;
```

Related Examples

- "Find Potential Data Type Issues in Generated Code" on page 78-5

Find Potential Data Type Issues in Generated Code

In this section...

Data Type Issues Overview

When you generate C code from MATLAB code, you can highlight potential data type issues in the C code generation report. The report highlights MATLAB code that requires single-precision, double-precision, or expensive fixed-point operations. The expensive fixed-point operations checks require a Fixed-Point Designer license.

- The double-precision check highlights expressions that result in a double-precision operation. When trying to achieve a strict-single or fixed-point design, manual inspection of code can be time-consuming and error prone.

  For a strict-single precision design, specify a standard math library that supports single-precision implementations. To change the library for a project, during the Generate Code step, in the project settings dialog box, on the **Custom Code** tab, set the **Standard math library** to `C99 (ISO)`.
- The single-precision check highlights expressions that result in a single operation.
- The expensive fixed-point operations check identifies optimization opportunities for fixed-point code. It highlights expressions in the MATLAB code that require cumbersome multiplication or division, expensive rounding, expensive comparison, or multiword operations. For more information on optimizing generated fixed-point code, see "Tips for Making Generated Code More Efficient" (Fixed-Point Designer).

Enable Highlighting of Potential Data Type Issues

Procedure 78.1. Enable the highlight option using the MATLAB Coder app

1. On the **Generate Code** page, to open the **Generate** dialog box, click the **Generate** arrow  .
2. Set **Build type** to one of the following:
   - `Source Code`
   - `Static Library (.lib)`
   - `Dynamic Library (.dll)`
   - `Executable (.exe)`
3. Click **More Settings**.
4. On the **Debugging** tab, select the **Always create a code generation report** and **Highlight potential data type issues** check boxes.

Procedure 78.2. Enable the highlight option using the command-line interface

1. Create an embedded code configuration object for `'lib'`, `'dll'`, or `'exe'`:

   ```
   cfg = coder.config('lib','ecoder',true); % or dll or exe
   ```
2. Set the `GenerateReport` and `HighlightPotentialDataTypeIssues` configuration object properties to `true`:

   ```
   cfg.GenerateReport = true;
   cfg.HighlightPotentialDataTypeIssues = true;
   ```

Find and Address Cumbersome Operations

Cumbersome operations usually occur due to an insufficient range of output. Avoid inputs to a multiply or divide operation that have word lengths larger than the base integer type of your processor. Software can process operations with larger word lengths, but this approach requires more code and runs slower.

This example requires Embedded Coder and Fixed-Point Designer licenses to run. The target word length for the processor in this example is 64.

1. Create the function `myMul`.

   ```
   function out = myMul(in1, in2)
       out = fi(in1*in2, 1, 64, 0);
   end
   ```
2. Generate code for `myMul`.

   ```
   cfg = coder.config('lib');
   ```

```
cfg.GenerateReport = true;
cfg.HighlightPotentialDataTypeIssues = true;
fm = fimath('ProductMode', 'SpecifyPrecision', 'ProductWordLength', 64);
codegen -config cfg myMul -args {fi(1, 1, 64, 4, fm), fi(1, 1, 64, 4, fm)}
```

3. Click **View report**.
4. In the Code Generation Report, on the left pane, click the **MATLAB code** tab.
5. Expand the **Highlight** section and select the **Expensive fixed-point operations** check box.

To resolve this issue, modify the data types of `in1` and `in2` so that the word length of the product does not exceed the target word length of 64.

Find and Address Expensive Rounding

Traditional handwritten code, especially for control applications, almost always uses "no effort" rounding. For example, for unsigned integers and two's complement signed integers, shifting right and dropping the bits is equivalent to rounding to floor. To get results comparable to, or better than, what you expect from traditional handwritten code, use the `floor` rounding method.

This example requires Embedded Coder and Fixed-Point Designer licenses to run.

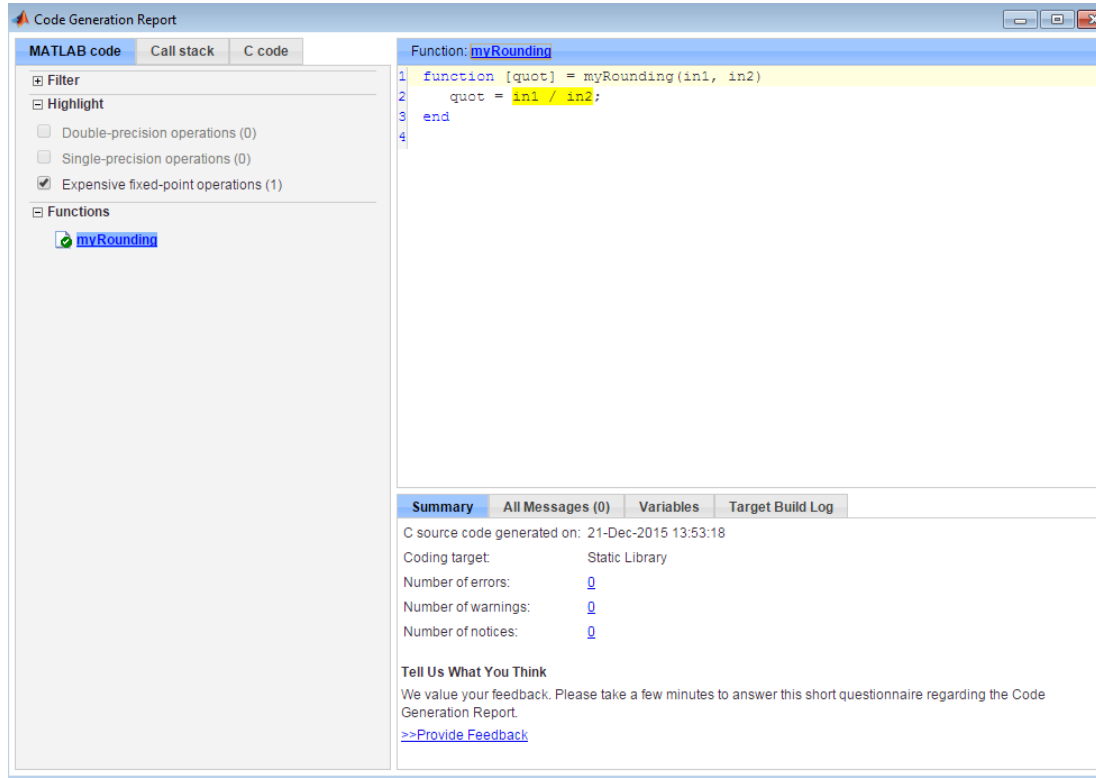
1. Create the function `myRounding`.

```
function [quot] = myRounding(in1, in2)
   quot = in1 / in2;
end
```

2. Generate code for `myRounding`.

```
cfg = coder.config('lib');
cfg.GenerateReport = true;
cfg.HighlightPotentialDataTypeIssues = true;
codegen -config cfg myRounding -args {fi(1, 1, 16, 2), fi(1, 1, 16, 4)}
```

3. Click **View report**.
4. In the Code Generation Report, on the left pane, click the **MATLAB code** tab.
5. Expand the **Highlight** section and select the **Expensive fixed-point operations** check box.



This division operation uses the default rounding method, `nearest`. Changing the rounding method to `Floor` provides a more efficient implementation.

Find and Address Expensive Comparison Operations

Comparison operations generate extra code when a casting operation is required to do the comparison. For example, before comparing an unsigned integer to a signed integer, one of the inputs must be cast to the signedness of the other. Consider optimizing the data types of the input arguments so that a cast is not required in the generated code.

This example requires Embedded Coder and Fixed-Point Designer licenses to run.

1 Create the function `myRelop`.

```
function out = myRelop(in1, in2)
    out = in1 > in2;
end
```

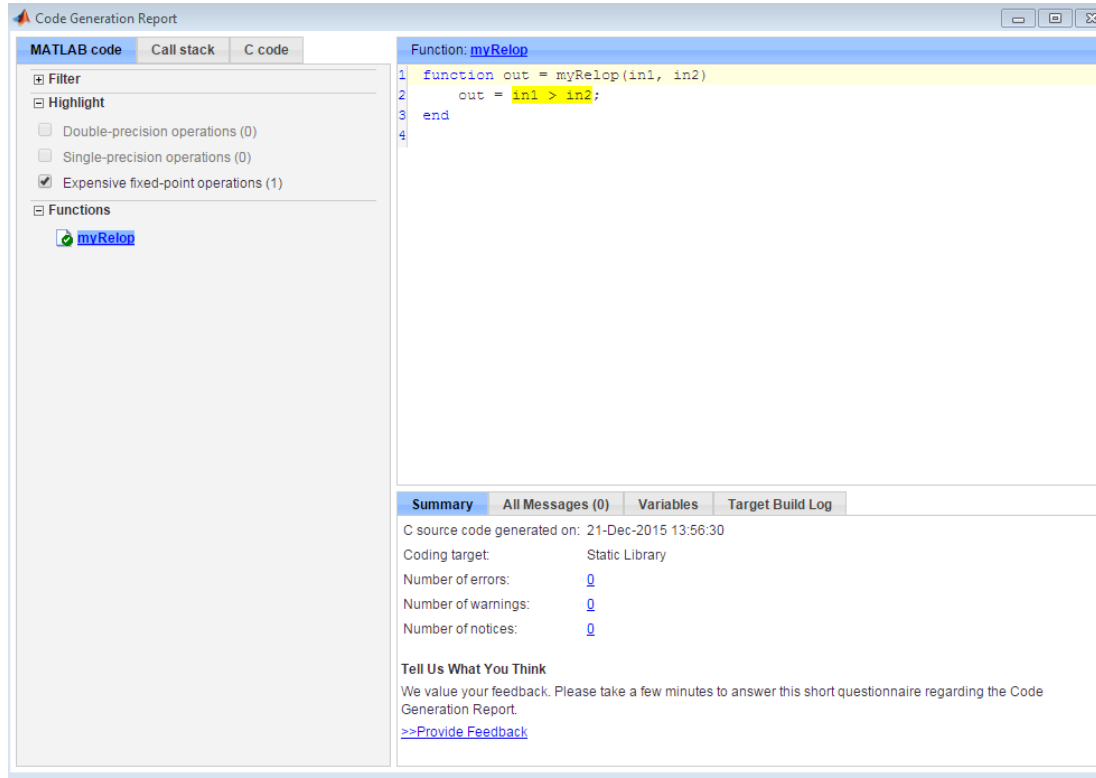
2 Generate code for `myRelop`.

```
cfg = coder.config('lib');
cfg.GenerateReport = true;
cfg.HighlightPotentialDataTypeIssues = true;
codegen -config cfg myRelop -args {fi(1, 1, 14, 3, 1), fi(1, 0, 14, 3, 1)}
```

3 Click **View report**.

4 In the Code Generation Report, on the left pane, click the **MATLAB code** tab.

5 Expand the **Highlight** section and select the **Expensive fixed-point operations** check box.



The first input argument, `in1`, is signed, while `in2` is unsigned. Extra code is generated because a cast must occur before the two inputs can be compared.

Change the signedness and scaling of one of the inputs to generate more efficient code.

Find and Address Multiword Operations

Multiword operations can be inefficient on hardware. When an operation has an input or output data type larger than the largest word size of your processor, the generated code contains multiword operations. You can avoid multiword operations in the generated code by specifying local `fimath` properties for variables. You can also manually specify input and output word lengths of operations that generate multiword code.

This example requires Embedded Coder and Fixed-Point Designer licenses to run. The target word length is 64 in this example.

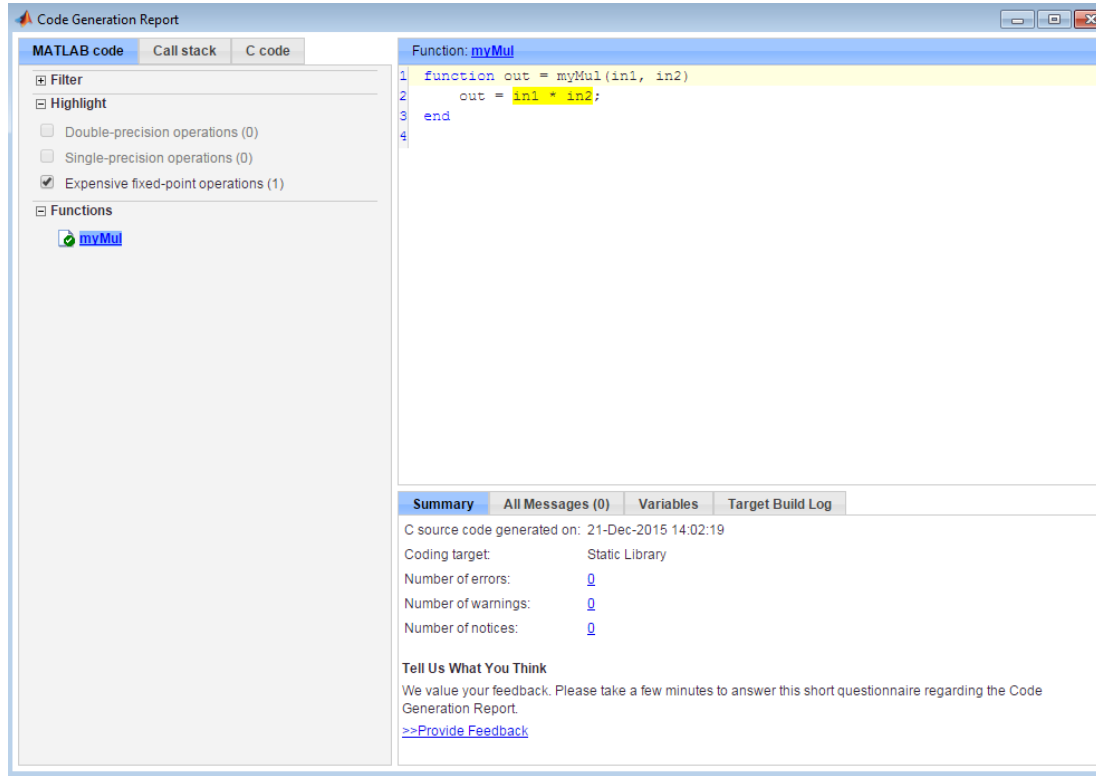
1. Create the function `myMul`.

```
function out = myMul(in1, in2)
    out = in1 * in2;
end
```

2. Generate code for `myMul`.

```
cfg = coder.config('lib');
cfg.GenerateReport = true;
cfg.HighlightPotentialDataTypeIssues = true;
codegen -config cfg myMul -args {fi(1, 1, 33, 4), fi(1, 1, 32, 4)}
```

3. Click **View report**.
4. In the Code Generation Report, on the left pane, click the **MATLAB code** tab.
5. Expand the **Highlight** section and select the **Expensive fixed-point operations** check box.



The `in1 * in2` operation is highlighted in the HTML report. On the bottom pane, click the **Variables** tab. The word length of `in1` is 33 bits, and the word length of `in2` is 32 bits. Hovering over the highlighted expression reveals that the product has a word length of 65, which is larger than the target word length of 64. Therefore, the software detects a multiword operation.

To resolve this issue, modify the data types of `in1` and `in2` so the word length of the product does not exceed the target word length, or specify the `ProductMode` property of the local `fimath` object.

PIL Execution with ARM Cortex-A at the Command Line

This example shows how to set up a PIL execution to verify generated code at the command line.

You can use processor-in-the-loop (PIL) executions to verify generated code that you deploy to target hardware by using a MATLAB Coder procedure. You can profile algorithm performance and speed for your generated code. To verify generated code with the MATLAB Coder app, you must have an Embedded Coder license.

This PIL execution is available with these hardware support packages. To use the PIL execution, install one of these support packages.

- Embedded Coder Support Package for BeagleBone Black Hardware
- Embedded Coder Support Package for ARM Cortex-A Processors

In the Command Window, select the hardware for PIL execution.

```
hw = coder.hardware('ARM Cortex-A9 (QEMU)')
```

```
hw =

  Hardware with properties:

            Name: 'ARM Cortex-A9 (QEMU)'
    CPUClockRate: 1000
```

When using the BeagleBone hardware, more hardware properties are supported (`Username`, `Password`, and `DeviceAddress`). Set these properties based on your specific hardware or application.

```
hw = coder.hardware('BeagleBone Black')
```

```
hw =

  Hardware with properties:

              Name: 'BeagleBone Black'
     CPUClockRate: 1000
         Password: 'root'
         Username: 'admin'
    DeviceAddress: '192.168.1.10'
```

Add the hardware to the MATLAB Coder configuration object.

```
cfg = coder.config('lib','ecoder',true);
cfg.VerificationMode = 'PIL';
cfg.Hardware = hw;
```

Generate PIL code for a function, `averaging_filter`.

```
codegen -config cfg averaging_filter -args {zeros(1,16)}
```

For more information on the `averaging_filter` function, see the “Averaging Filter” example in “MATLAB Coder Examples” (MATLAB Coder).

For another example of PIL verification, see the "Processor-in-the-Loop Verification of MATLAB Functions" page in the documentation of the Embedded Coder Support Package for ARM Cortex-A Processors. To install the Embedded Coder Support Package for ARM Cortex-A Processors, see “Supported Hardware”.

PIL Execution with ARM Cortex-A by Using the MATLAB Coder App

You can use processor-in-the-loop (PIL) executions to verify generated code that you deploy to target hardware by using a MATLAB Coder procedure. You can profile algorithm performance and speed for your generated code. To verify generated code with the MATLAB Coder app, you must have an Embedded Coder license.

This PIL execution is available with these hardware support packages. To use the PIL execution, install one of these support packages.

- Embedded Coder Support Package for BeagleBone Black Hardware
- Embedded Coder Support Package for ARM Cortex-A Processors

You can set up PIL execution with the MATLAB Coder app.

To configure the build type and hardware board:

1. On the **Generate Code** page, in the **Generate** dialog box:
   - Set the **Build type** to `Static Library`.
   - Clear the **Generate code only** check box.
   - Set the **Hardware Board** to `BeagleBone Black` or `ARM Cortex-A9 (QEMU)`.
2. If necessary, modify the settings for your board. To modify the settings, click **More Settings**, and then click **Hardware**.
3. To generate the library, click **Generate**.
4. Set up your PIL execution. Click **Verify Code** to open the **Verify Code** dialog box.

   Because the hardware board is not `MATLAB Host Computer`, the **Verify Code** dialog box is configured for PIL execution.

   In the **Verify Code** dialog box:
   - Enter the name of the test file to use for PIL execution.
   - Select **Generated code**.
5. To start the PIL execution, click **Run Generated Code**.
6. To stop the PIL execution, click **Stop**.

For another example of PIL verification, see the "Processor-in-the-Loop Verification of MATLAB Functions" page in the documentation of the Embedded Coder Support Package for ARM Cortex-A Processors. Install the support package to view the documentation.