

Real-Time Workshop[®]

For Use with Simulink[®]

- Modeling
- Simulation
- Implementation

How to Contact The MathWorks:



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



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



508-647-7000	Phone
--------------	-------



508-647-7001	Fax
--------------	-----



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

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

Real-Time Workshop® User's Guide

© COPYRIGHT 1994 - 2004 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

MATLAB, Simulink, Stateflow, Handle Graphics, and Real-Time Workshop are registered trademarks, and TargetBox is a trademark of The MathWorks, Inc.

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

Printing History:	May 1994	First printing	Version 1
	January 1998	Second printing	Version 2.1
	January 1999	Third printing	Version 3.11 (Release 11)
	September 2000	Fourth printing	Version 4 (Release 12)
	June 2001	Online only	Updated for Version 4.1 (Release 12.1)
	October 2001	Online only	Updated for Version 4.2 (Release 13)
	July 2002	Online only	Updated for Version 5.0 (Release 13)
	June 2004	Online only	Updated for Version 6.0 (Release 14)

Understanding Real-Time Workshop

1

Product Overview	1-2
Some Real-Time Workshop Capabilities	1-3
Software Design with Real-Time Workshop	1-3
The Rapid Prototyping Process	1-5
Key Aspects of Rapid Prototyping	1-5
Rapid Prototyping for Digital Signal Processing	1-8
Rapid Prototyping for Control Systems	1-9
Open Architecture of Real-Time Workshop	1-11
Where to Find Help	1-14
How Do I...	1-14

Code Generation and the Build Process

2

The Real-Time Workshop User Interface	2-2
Using the Real-Time Workshop Pane	2-2
Target, Documentation, and Build Options	2-4
Simulation Parameters and Code Generation	2-11
Solver Options	2-11
Data Import and Export Options	2-13
Optimization Options	2-17
Diagnostics Options	2-26
Hardware Implementation Options	2-28
Model Referencing Options	2-34
Real-Time Workshop Options	2-35
Tracing Generated Code Back to Your Simulink Model	2-45

Other Interactions Between Simulink and Real-Time Workshop	2-46
Selecting a Target Configuration	2-53
The System Target File Browser	2-53
Available Targets	2-54
Making an Executable	2-59
Construction of Symbols	2-59
Generated Source Files and File Dependencies	2-60
Compilation and Linking	2-66
Choosing and Configuring Your Compiler	2-68
Choosing and Configuring Your Compiler on Windows	2-68
Choosing and Configuring Your Compiler on UNIX	2-70
Including S-function Source Code	2-70
Template Makefiles and Make Options	2-71
Compiler-Specific Template Makefiles	2-71
Configuring the Generated Code with TLC	2-77
Target Language Compiler Variables and Options	2-77

Generated Code Formats

3

Introduction	3-2
Targets and Code Formats	3-2
Choosing a Code Format for Your Application	3-9
Real-Time Code Format	3-13
Unsupported Blocks	3-13
System Target Files	3-13
Template Makefiles	3-13
Real-Time malloc Code Format	3-15

Unsupported Blocks	3-15
System Target Files	3-15
Template Makefiles	3-16

S-Function Code Format	3-17
-------------------------------------	-------------

Embedded C Code Format	3-18
Making GRT-Based Targets ERT-Compatible	3-18

4

Building Subsystems and Working with Referenced Models

Nonvirtual Subsystem Code Generation	4-2
Nonvirtual Subsystem Code Generation Options	4-2
Modularity of Subsystem Code	4-14
Code Reuse Diagnostics	4-14

Generating Code and Executables from Subsystems	4-15
--	-------------

Generating Code from Models Containing Model Blocks .	4-18
About Model Reference	4-18
Using Referenced Models	4-21
Project Directory Structure for Model Reference Targets	4-27
Supporting the Shared Utilities Directory in the Build Process	4-33
Inherited Sample Time for Referenced Models	4-35
Reusing Code for Subsystems in Referenced Models	4-40
Making Custom Targets Model Reference Compatible	4-42

5

Working with Data Structures

Parameters: Storage, Interfacing, and Tuning	5-2
---	------------

Storage of Nontunable Parameters	5-2
Tunable Parameter Storage	5-4
Storage Classes of Tunable Parameters	5-6
Using the Model Parameter Configuration Dialog	5-8
Tunable Expressions	5-13
Tunability of Linear Block Parameters	5-16
Parameter Configuration Quick Reference Diagram	5-18
Signal Storage, Optimization, and Interfacing	5-19
Signal Storage Concepts	5-19
Signals with Auto Storage Class	5-22
Declaring Test Points	5-25
Interfacing Signals to External Code	5-27
Symbolic Naming Conventions for Signals in Generated Code	5-29
Summary of Signal Storage Class Options	5-30
C API for Parameter Tuning and Signal Monitoring	5-31
Target Language Compiler API for Parameter	
Tuning and Signal Monitoring	5-31
Parameter Tuning via MATLAB Commands	5-31
Simulink Data Objects and Code Generation	5-33
Parameter Objects	5-34
Signal Objects	5-38
Resolving Conflicts in Configuration of Parameter	
and Signal Objects	5-42
Customizing Code for Parameter and Signal Objects	5-46
Using Objects to Export ASAP2 Files	5-46
Block States: Storing and Interfacing	5-47
Storage of Block States	5-47
Block State Storage Classes	5-48
Using the State Properties Dialog to Interface	
States to External Code	5-49
Symbolic Names for Block States	5-50
Block States and Simulink Signal Objects	5-53
Summary of State Storage Class Options	5-55
Storage Classes for Data Store Memory Blocks	5-56
Data Store Memory and Simulink Signal Objects	5-58

6

Introduction 6-2

Using the External Mode User Interface 6-3

- External Mode Interface Options 6-3
- External Mode Related Menu and Toolbar Items 6-5
- External Mode Control Panel 6-10
- Target Interfacing 6-12
- External Signal Uploading and Triggering 6-13
- Data Archiving 6-18
- Parameter Downloading 6-20

External Mode Compatible Blocks and Subsystems 6-22

- Compatible Blocks 6-22
- Signal Viewing Subsystems 6-22

External Mode Communications Overview 6-25

- The Download Mechanism 6-25
- Inlined and Tunable Parameters 6-26

Client/Server Implementations 6-28

- Using the TCP/IP Implementation 6-28
- Using the Serial Implementation 6-30
- Running the External Program 6-31
- Implementing an External Mode Protocol Layer 6-34

Limitations of External Mode 6-35

- Error Conditions 6-35

Program Architecture

7

Introduction 7-2

Model Execution 7-4

Models for Non-Real-Time Single-Tasking Systems	7-5
Models for Non-Real-Time Multitasking Systems	7-6
Models for Real-Time Single-Tasking Systems	7-7
Models for Real-Time Multitasking Systems	7-9
Models for Multitasking Systems that Use Real-Time Tasking Primitives	7-11
Program Timing	7-12
Program Execution	7-13
External Mode Communication	7-14
Data Logging in Single Tasking and Multitasking Model Execution	7-14
Rapid Prototyping and Embedded Model Execution Differences	7-15
Rapid Prototyping Model Functions	7-16
Embedded Model Functions	7-21
Rapid Prototyping Program Framework	7-23
Rapid Prototyping Program Architecture	7-23
Rapid Prototyping System-Dependent Components	7-25
Rapid Prototyping System-Independent Components	7-26
Rapid Prototyping Application Components	7-29
Embedded Program Framework	7-36

Models with Multiple Sample Rates

8

Introduction	8-2
Single-tasking vs. Multitasking Environments	8-3
Executing Multitasking Models	8-5
Multitasking and Pseudomultitasking	8-6
Building the Program for Multitasking Execution	8-8
Single-tasking	8-8
Building the Program for Single-tasking Execution	8-9
Model Execution	8-9
Simulating Models with Simulink	8-10

Executing Models in Real Time	8-10
Single-tasking vs. Multitasking Operation	8-11
Sample Rate Transitions	8-12
Data Transfer Problems	8-13
Rate Transition Block Options	8-14
Faster to Slower Transitions in Simulink	8-17
Faster to Slower Transitions in Real Time	8-18
Slower to Faster Transitions in Simulink	8-20
Slower to Faster Transitions in Real Time	8-20
Single-tasking and Multitasking	
Execution of a Model: an Example	8-24
Single-tasking Execution	8-25
Multitasking Execution	8-27

Optimizing the Model for Code Generation

9

Optimization Parameters Overview	9-2
Optimizing Models	9-4
Using the Model Advisor	9-4
Demos Illustrating Optimizations	9-6
Other Optimization Tools and Techniques	9-7
Expression Folding	9-10
Expression Folding Example	9-10
Using and Configuring Expression Folding	9-12
Conditional Input Execution	9-16
Block Diagram Performance Tuning	9-17
Lookup Tables and Polynomials	9-17
Accumulators	9-28
Use of Data Types	9-30
Additional Integer and Fixed-Point Optimizations	9-34

Introduction	10-2
Classes of Problems Solved by S-Functions	10-2
Types of S-Functions	10-3
Basic Files Required for Implementation	10-5
Noninlined S-Functions	10-7
S-Function Module Names for Real-Time Workshop Builds ..	10-7
Writing Wrapper S-Functions	10-9
MEX S-Function Wrapper	10-9
TLC S-Function Wrapper	10-13
The Inlined Code	10-17
Fully Inlined S-Functions	10-18
Multiport S-Function Example	10-18
Fully Inlined S-Function with the mdlRTW Routine	10-20
S-Function RTWdata	10-21
The Direct-Index Lookup Table Algorithm	10-22
The Direct-Index Lookup Table Example	10-23
Supporting Expression Folding in S-Functions	10-46
Categories of Output Expressions	10-47
Acceptance or Denial of Requests for	
Input Expressions	10-52
Utilizing Expression Folding in Your TLC Block	
Implementation	10-55
Creating Code-Reuse-Compatible S-Functions	10-61
Coding S-functions for Multirate	
Multitasking Environments	10-62
Rate Grouping Support in S-Functions	10-62
Creating Multitasking-Safe, Multirate,	
Port-Based Sample Time S-Functions	10-63

Introduction	11-2
Intellectual Property Protection	11-3
Creating an S-Function Block from a Subsystem	11-4
Sample Time Propagation in Generated S-Functions	11-8
Choice of Solver Type	11-8
Sample Time Propagation in Generated S-Functions	11-9
Choice of Solver Type	11-9
Tunable Parameters in Generated S-Functions	11-10
Automated S-Function Generation	11-12
Restrictions	11-16
Limitations on Use of Goto and From Blocks	11-16
Model Reference and Updating Restrictions	11-17
Unsupported Blocks	11-19
System Target File and Template Makefiles	11-20
System Target File	11-20
Template Makefiles	11-20

Real-Time Workshop Rapid Simulation Target

Introduction	12-2
Licensing Protocols for Simulink Solvers in Executables	12-3
Building for the Rapid Simulation Target	12-5
Configuration Options for RSim Targets	12-6
Running a Rapid Simulation	12-7
Simulation Performance	12-17
Batch and Monte Carlo Simulations	12-17

Limitations 12-18

Targeting Tornado for Real-Time Applications

13

The Tornado Environment 13-2

- Confirming Your Tornado Setup Is Operational 13-2
- VxWorks Library 13-3

Run-Time Architecture Overview 13-5

- Parameter Tuning and Monitoring 13-5

Implementation Overview 13-12

- Adding Device Driver Blocks 13-14
- Configuring the Template Makefile 13-14
- Tool Locations 13-15
- Building the Program 13-15
- Downloading and Running the Executable Interactively 13-20

Custom Code Blocks

14

Introduction 14-2

Custom Code Library 14-4

- Custom Code in Subsystems 14-8

Timing Services

15

Absolute and Elapsed Time Computation 15-2

Timers for Periodic and Asynchronous Tasks	15-2
Allocation of Timers	15-3
Integer Timers in Generated Code	15-3
Elapsed Time Counters in Triggered Subsystems	15-3
C and TLC APIs for Accessing Timers	15-5
Elapsed Timer Code Generation Example	15-9

Asynchronous Support

16

Introduction	16-2
VxWorks Library Overview	16-2
Accessing the VxWorks Library	16-3
Generating Code with the VxWorks Library Blocks	16-4
Demos and Additional Information	16-4
Interrupt Handling Blocks	16-5
Async Interrupt Block	16-5
Task Synchronization Block	16-15
Rate Transitions and Asynchronous Blocks	16-23
Handling Rate Transitions Between Asynchronous and/or Periodic Tasks	16-24
Handling Multiple Asynchronous Interrupts	16-26
Using Timers in Asynchronous Tasks	16-29
Creating a Customized Asynchronous Library	16-31
Async Interrupt Block Implementation	16-31
Task Synchronization Block Implementation	16-35
asynclib.tlc Support Library	16-36

17

- C-API for Interfacing with Signals and Parameters 17-2**
 - Generating the C-API Files 17-3
 - Description of C-API Files 17-5
 - Using the C-API in an Application 17-12
 - C-API Limitations 17-15
 - Target Language Compiler API for
Signals and Parameters 17-16

- Creating an External Mode Communication Channel . . . 17-18**
 - The Design of External Mode 17-18
 - External Mode Communications Overview 17-21
 - External Mode Source Files 17-22
 - Guidelines for Implementing the Transport Layer 17-25

- Combining Multiple Models 17-29**
 - Using GRT Malloc to Combine Models 17-30

Blocks That Depend on Absolute Time

A

Generating ASAP2 Files

B

- Overview B-2**
- Targets Supporting ASAP2 B-3**
- Defining ASAP2 Information B-4**
- Generating an ASAP2 File B-6**

Customizing an ASAP2 File B-12

Structure of the ASAP2 File B-18

The Real-Time Workshop Development Process

C

Introduction C-2

A Next-Generation Development Tool C-3

 Key Features C-4

 Benefits C-7

 Integration with Simulink C-10

How MathWorks Tools Streamline Development C-12

Code Formats C-18

 Target Environments C-21

 Code Generation Optimizations C-27

An Open and Extensible Environment C-31

Glossary

Index

Understanding Real-Time Workshop

The following sections describe the architecture and application of Real-Time Workshop, and summarize the contents and organization of its documentation, giving an overview of its contents and some entry points into it for a number of topics of interest.

Product Overview (p. 1-2)

Real-Time Workshop at a glance

The Rapid Prototyping Process (p. 1-5)

Key advantages of rapid prototyping, along with descriptions of its application in two domains

Open Architecture of Real-Time Workshop (p. 1-11)

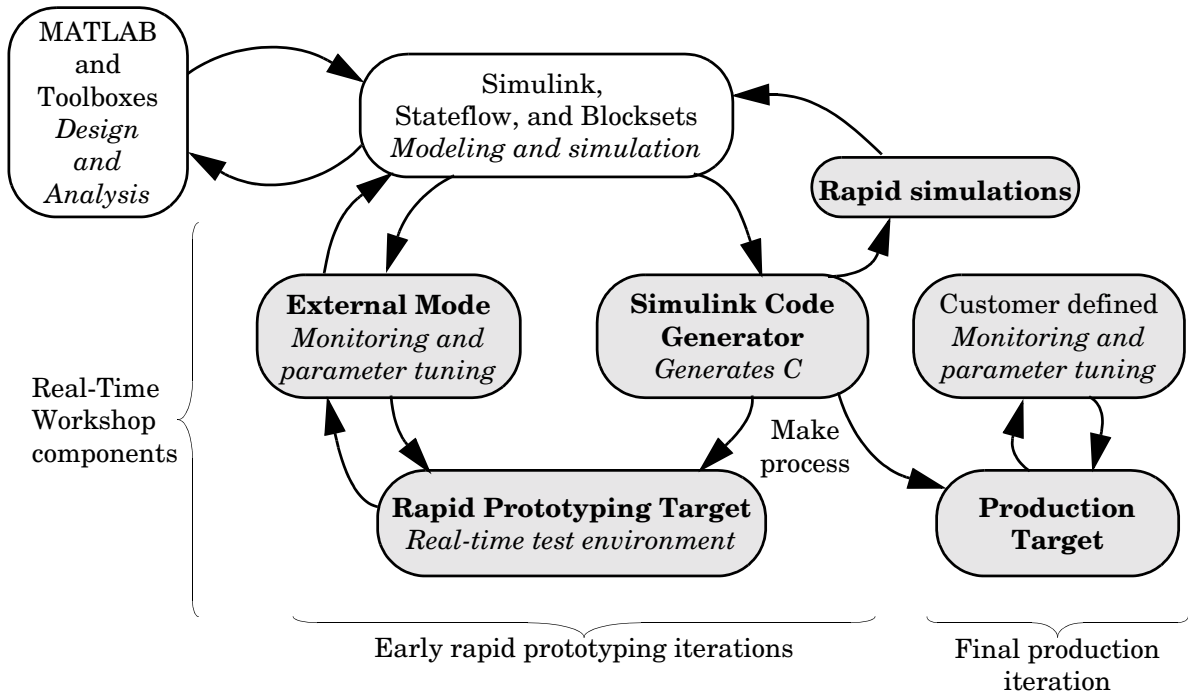
Modules and files involved in code generation that you can customize for your own targets and applications

Where to Find Help (p. 1-14)

Pointers to both basic descriptions and advanced information on specific topics

Product Overview

Real-Time Workshop[®] generates optimized, portable, and customizable ANSI C code from Simulink[®] models to create stand-alone implementations of models that operate in real-time and non-real-time in a variety of target environments. Generated code can run on PC hardware, DSPs, microcontrollers on bare-board environments, and with commercial or proprietary real-time operating systems (RTOS). Real-Time Workshop lets you speed up simulations, build in intellectual property protection, and operate across a wide variety of real-time rapid prototyping targets. The figure below illustrates the role of Real-Time Workshop (shaded elements) in the software design process.



Software Design and Deployment Using MATLAB and Simulink

Some Real-Time Workshop Capabilities

With Real-Time Workshop, you can quickly generate code for discrete-time, continuous-time (fixed-step), and hybrid systems, as well as for finite state machines modeled in Stateflow[®] using the optional Stateflow Coder. The optional Real-Time Workshop Embedded Coder works with Real-Time Workshop to generate efficient, embeddable source code. These core products support a growing set of embedded targets, such as Embedded Target for Motorola[®] MPC555, Embedded Target for the TI C6000[™] DSP Platform, and the Embedded Target for OSEK/VDX operating environments.

Using integrated makefile-based targetting support, Real-Time Workshop builds programs that can help speed up your simulations, provide intellectual property protection, and run on a wide variety of real-time rapid prototyping or production targets. Simulink's external mode run-time monitor works seamlessly with real-time targets, providing an elegant signal monitoring and parameter tuning interface. Real-Time Workshop supports continuous-time, discrete-time, and hybrid systems, including conditionally executed and atomic systems. Real-Time Workshop accelerates your development cycle, producing higher quality results in less time.

Real-Time Workshop is a key link in the set of system design tools provided by The MathWorks, providing a real-time development environment — a direct path from system design to hardware implementation. You can streamline application development and reduce costs with Real-Time Workshop by testing design iterations with real-time hardware. Real-Time Workshop supports the execution of dynamic system models on hardware by automatically converting models to code and providing model-based debugging support. It is well suited for accelerating simulations, rapid prototyping, turnkey solutions, and production of embedded real-time applications.

Software Design with Real-Time Workshop

A typical product cycle using MathWorks tools starts with modeling in Simulink, followed by an analysis of the simulations in MATLAB[®]. During the simulation process, you use the rapid simulation features of Real-Time Workshop to speed up your simulations.

After you are satisfied with the simulation results, you use Real-Time Workshop in conjunction with a rapid prototyping target, such as xPC Target. The rapid prototyping target is connected to your physical system. You test and observe your system, using your Simulink model as the interface to your

physical target. Once your simulation is functioning properly, you use Real-Time Workshop to transform your model to C code. An extensible make process and download procedure creates an executable for your model and places it on the target system. Finally, using external mode, you can monitor and tune parameters in real-time as your model executes on the target environment.

There are two broad classes of targets: rapid prototyping targets and the embedded target. Code generated for the rapid prototyping targets supports increased monitoring and tuning capabilities. Code generated for embedded targets is highly optimized and suitable for deployment in production systems, and can include application-specific entry points to monitor signals and tune parameters.

To support embedded targets, The MathWorks distributes Real-Time Workshop Embedded Coder as a separate product. Embedded Coder is an extension of Real-Time Workshop designed to generate C code for embedded discrete-time systems, where efficiency, configurability, readability, and traceability of the generated code are extremely important. Real-Time Workshop Embedded Coder enhances Real-Time Workshop code generation technology to generate embeddable ANSI C or ISO C code that compares favorably with hand-optimized code in terms of performance, ROM code size, RAM requirements, and readability. The Real-Time Workshop Embedded Coder documentation contains information about optimization specifically for embedded code.

For a more complete general overview of the key features, capabilities, and benefits of Real-Time Workshop, see Appendix C, “The Real-Time Workshop Development Process.”

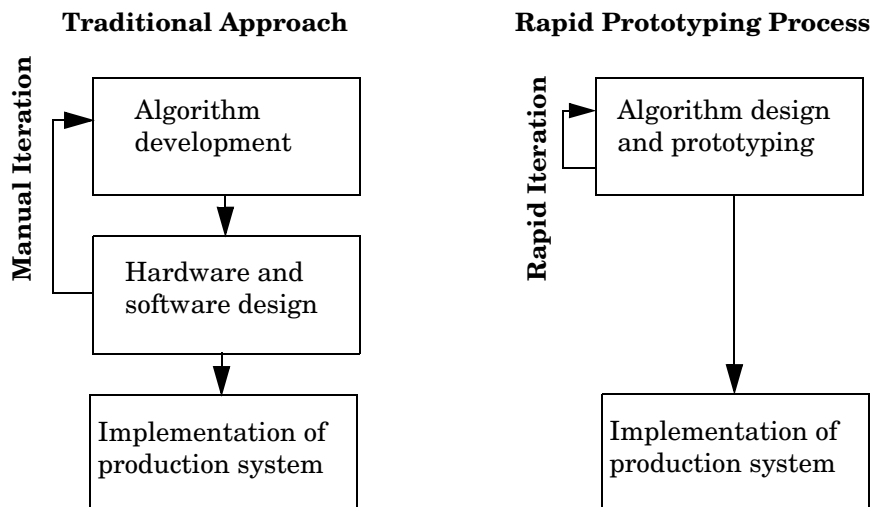
The Rapid Prototyping Process

Real-Time Workshop supports *rapid prototyping*, an application development process that allows you to

- Conceptualize solutions graphically in a block diagram modeling environment
- Evaluate system performance early on—before laying out hardware, coding production software, or committing to a fixed design
- Refine your design by rapid iteration between algorithm design and prototyping
- Tune parameters while your real-time model runs, using Simulink in external mode as a graphical front end

Key Aspects of Rapid Prototyping

The figure below contrasts the rapid prototyping development process with the traditional development process.



Traditional vs. Rapid Prototyping Development Processes

The traditional approach to real-time design and implementation typically involves multiple teams of engineers, including an algorithm design team, software design team, hardware design team, and an implementation team. When the algorithm design team has completed its specifications, the software design team implements the algorithm in a simulation environment and then specifies the hardware requirements. The hardware design team then creates the production hardware. Finally, the implementation team integrates the hardware into the larger overall system.

This traditional development process takes so much time because algorithm designers often do not have access to the hardware that is actually deployed. The rapid prototyping process combines the algorithm, software, and hardware design phases, eliminating potential bottlenecks by allowing engineers to see results and rapidly iterate solutions before building expensive hardware.

Automating Programming

Automatic program building allows you to make design changes directly to the block diagram, putting algorithm development (including coding, compiling, linking, and downloading to target hardware) under control of a single process:

- **Design a Model in Simulink**

You begin the rapid prototyping process with the development of a model in Simulink. In control engineering, you model plant dynamics and other dynamic components that constitute a controller and/or an observer.

- **Simulate your Model in Simulink**

You use MATLAB, Simulink, and toolboxes to aid in the development of algorithms and analysis of the results. If the results are not satisfactory, you can iterate the modeling and analysis process until results are acceptable.

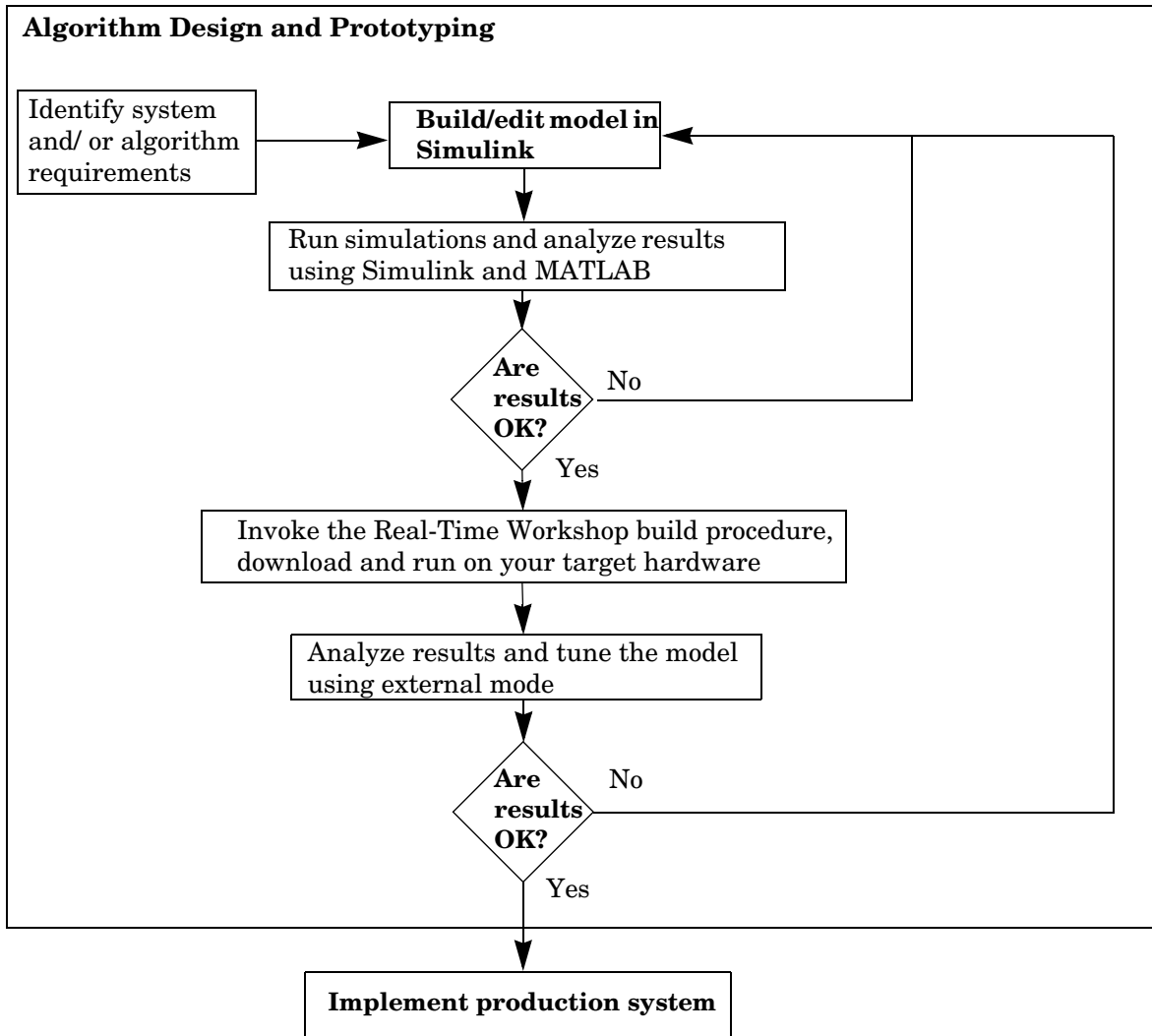
- **Generate Source Code with Real-Time Workshop**

Once simulation results are acceptable, you generate downloadable C code that implements the appropriate portions of the model. You can use Simulink in external mode to tune parameters and further refine your model, quickly iterating through solutions.

- **Implement a Production Prototype**

At this stage, the rapid prototyping process is complete. You can begin the final implementation for production with confidence that the underlying algorithms work properly in your real-time production system.

The next figure illustrates the flow of this process.



The Rapid Prototyping Development Process

Highly productive development cycles are possible due to the integration of Real-Time Workshop, MATLAB, and Simulink. Each component adds value to your application design process:

- **MATLAB:** Provides design, analysis, and data visualization tools.
- **Simulink:** Provides system modeling, simulation, and validation.
- **Real-Time Workshop:** Generates C code from Simulink model; provides framework for running generated code in real-time, tuning parameters, and viewing real-time data.

Rapid Prototyping for Digital Signal Processing

The first step in the rapid prototyping process for digital signal processing is to consider the kind and quality of the data to be worked on, and to relate it to the system requirements. Typically this includes examining the signal-to-noise ratio, distortion, and other characteristics of the incoming signal, and relating them to algorithm and design choices.

System Simulation and Algorithm Design

In the rapid prototyping process, the block diagram plays two roles in algorithm development. The block diagram helps to identify processing bottlenecks, and to optimize the algorithm or system architecture. The block diagram also functions as a high-level system description. That is, the diagram provides a hierarchical framework for evaluating the behavior and accuracy of alternative algorithms under a range of operating conditions.

Analyzing Results, Tuning Parameters, and Monitoring Signals

After you create an algorithm (or a set of candidate algorithms), the next stage is to consider architectural and implementation issues. These include complexity, speed, and accuracy. In a conventional development environment, this would mean running the algorithm and recoding it in C or in a hardware design and simulation package.

Simulink external mode allows you to change parameters interactively while your signal processing algorithms execute in real time on the target hardware. After building the executable and downloading it to your hardware, you tune (modify) block parameters in Simulink. Simulink automatically downloads the new values to the hardware. You can monitor the effects of your parameter changes by simply connecting Scope blocks to signals that you want to observe.

Rapid Prototyping for Control Systems

Rapid prototyping for control systems is similar to digital signal processing, with one major difference. In control systems design, you must model your plant prior to developing algorithms in order to simulate closed-loop performance. Once your plant model is sufficiently accurate, the rapid prototyping process for control system design continues in much the same manner as digital signal processing design.

Rapid prototyping begins with developing block diagram plant models of sufficient fidelity for preliminary system design and simulation. Once simulations indicate acceptable system performance levels, the controller block diagram is separated from the plant model and I/O device driver blocks are attached to it. Automatic code generation immediately converts the entire system to real-time executable code, which can be automatically loaded onto target hardware.

Modeling Systems in Simulink

The first step in the design process is development of a plant model. The Simulink collection of linear and nonlinear components helps you to build models involving plant, sensor, and actuator dynamics. Because Simulink is customizable, you can further simplify modeling by creating custom blocks and block libraries from continuous- and discrete-time components.

Using the System Identification Toolbox, you can analyze test data to develop an empirical plant model; or you can use the Symbolic Math Toolbox to translate the equations of the plant dynamics into state-variable form.

Analyzing Simulation Results

You can use MATLAB and Simulink to analyze the results produced from a model developed in the first step of the rapid prototyping process. At this stage, you can design and add a controller to your plant.

Deriving and Analyzing Algorithms

From the block diagrams developed during the modeling stage, you can extract state-space models through linearization techniques. These matrices can be used in control system design. You can use the following tools to facilitate control system design, and work with the matrices that you derived:

- Control System Toolbox
- LMI Control Toolbox
- Model Predictive Control Toolbox
- Robust Control Toolbox
- System Identification Toolbox
- SimMechanics

Once you have your controller designed, you can create a closed-loop system by connecting it to the Simulink plant model. Closed-loop simulations allow you to determine how well the initial design meets performance requirements.

Once you have a satisfactory model, it is a simple matter to generate C code directly from the Simulink block diagram, compile it for the target processor, and link it with supplied or user-written application modules.

Analyzing Results, Tuning Parameters, and Monitoring Signals

You can load output data from your program into MATLAB for analysis, or display the data with third-party monitoring tools. You can easily make design changes to the Simulink model and then regenerate the C code.

Open Architecture of Real-Time Workshop

Real-Time Workshop is an open system designed for use with a wide variety of operating environments and hardware types. The figure “Real-Time Workshop Architecture” shows how you can extend key elements of Real-Time Workshop.

You can configure the Real-Time Workshop program generation process to your own needs by modifying the following components:

- Simulink and the model file (*model.mdl*)

Simulink provides a very high-level language (VHLL) development environment. The language elements are blocks and subsystems that visually embody your algorithms. You can think of Real-Time Workshop as a compiler that processes a VHLL source program (*model.mdl*), and emits code suitable for a traditional high-level language (HLL) compiler.

S-functions written in C let you extend the Simulink VHLL by adding new general purpose blocks, or incorporating legacy code into a block.

- The intermediate model description (*model.rtw*)

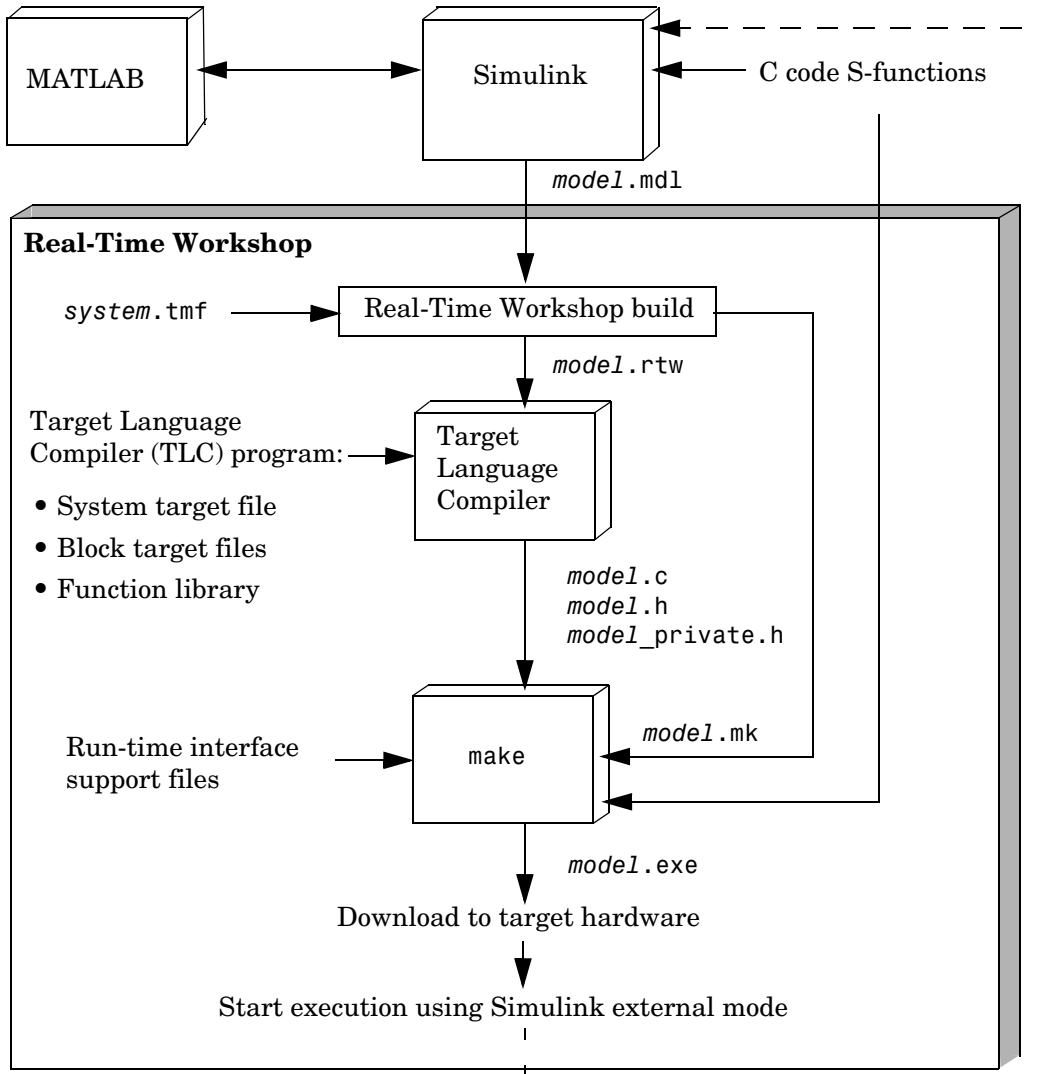
The initial stage of the code generation process is to analyze the source model. The resultant description file contains a hierarchical structure of records describing systems and blocks and their connections.

The S-function API includes a special function, `mdlRTW`, that lets you customize the code generation process by inserting parameter data from your own blocks into the *model.rtw* file.

- The Target Language Compiler (TLC) program

The Target Language Compiler interprets a program that reads the intermediate model description and generates code that implements the model as a program.

You can customize the elements of the TLC program in two ways. First, you can implement your own system target file, which controls overall code generation parameters. Second, you can implement block target files, which control how code is generated from individual blocks such as your own S-function blocks.



Real-Time Workshop Architecture

- Source code generated from the model; for descriptions of these files, see “Files and Directories Created by the Build Process” in the Real-Time Workshop Getting Started Guide.

There are several ways to customize generated code, or interface it to custom code:

- Exported entry points let you interface your hand-written code to the generated code. This makes it possible to develop your own timing and execution engine, or to combine code generated from several models into a single executable.
 - You can automatically make signals, parameters, and other data structures within generated code visible to your own code, facilitating parameter tuning and signal monitoring.
 - Prepare or modify Target Language Compiler script files to customize the transformation of Simulink blocks into source code. See the Target Language Compiler Reference Guide for further details.
- Run-time interface support files

The run-time interface consists of code interfacing to the generated model code. You can create a custom set of run-time interface files, including

- A harness (main) program
 - Code to implement a custom external mode communication protocol
 - Code that interfaces to parameters and signals defined in the generated code
 - Timer and other interrupt service routines
 - Hardware I/O drivers
- The template makefile and *model.mk*

A makefile, *model.mk*, controls the compilation and linking of generated code. Real-Time Workshop generates *model.mk* from a template makefile during the code generation and build process. You can create a custom template makefile to control compiler options and other variables of the make process.

All of these components contribute to the process of transforming a Simulink model into an executable program. The topics in the next section point you to documentation describing each of them.

Where to Find Help

Documentation for Real-Time Workshop and related products from The MathWorks covers many topics—some in considerable depth—and includes many examples of use. Some of the major topics covered are summarized below, enabling you to locate directly what you need to proceed.

If you are a less experienced user, you will benefit from reading the Getting Started Guide, which introduces the product and describes its capabilities, applications, benefits, and general usage. Inside that guide are tutorials that provide immediate hands-on experience to get you familiar with the look, feel, and capabilities of Real-Time Workshop. That guide also discusses

- The role of Real-Time Workshop in your development cycle
- Basic real-time system concepts and terms
- General and platform-specific installation instructions
- Related product descriptions
- Simulink demos that illustrate code generation

How Do I...

If you need specific details about how to use Real-Time Workshop, scan the topics and descriptions below to locate documentation relevant to your development tasks and interests. You can also search the index to find information not included in this list.

Operate the Real-Time Workshop User Interface

You control most aspects of code generation through the Real-Time Workshop tab of the **Configuration Parameters** dialog, and the dialogs descending from it. See “The Real-Time Workshop User Interface” on page 2-2 for full descriptions of the options at your disposal.

Select Targets and Customize Compilation

Setting up targets for code generation is simple with the Target File Browser, described in “Selecting a Target Configuration” on page 2-53. Look there also for information on configuring compilers (“Choosing and Configuring Your Compiler” on page 2-68) and modifying makefiles (“Template Makefiles and Make Options” on page 2-71). For details on working with specific targets, see Chapter 11, “The S-Function Target,” Chapter 12, “Real-Time Workshop Rapid

Simulation Target,” Chapter 13, “Targeting Tornado for Real-Time Applications,” Appendix B, “Generating ASAP2 Files”, and the Real-Time Workshop Embedded Coder documentation.

Generate Single- and Multitasking Code

Real-Time Workshop fully supports singletasking and multitasking code generation. See Chapter 7, “Program Architecture,” Chapter 8, “Models with Multiple Sample Rates,” Chapter 15, “Timing Services,” and Chapter 16, “Asynchronous Support” for a further details.

Customize Generated Code

Real-Time Workshop supports customization of the generated code.

You can include Custom Code blocks in any system of any model in order to insert comments, include directives and code fragments in specific functions. See Chapter 14, “Custom Code Blocks,” for further information. You can also insert C code via the **Custom Code** configuration set dialog.

The most flexible approach to customizing generated code is to modify Target Language Compiler (TLC) files. The Target Language Compiler is an interpreted language that translates Simulink models into C code. Using the Target Language Compiler, you can direct the code generation process.

There are two TLC files, `hookslib.tlc` and `cachelib.tlc`, that contain functions you can use to customize Real-Time Workshop generated code. See the Target Language Compiler documentation for details on these TLC files. See also the source code, located in `matlabroot/rtw/c/tlc/lib/cachelib.tlc` and `matlabroot/rtw/c/tlc/mw/hookslib.tlc`.

Optimize Generated Code

The default code generation settings are generic for flexible rapid prototyping systems. The penalty for this flexibility is code that is less than optimal. There are several optimization techniques that you can use to minimize the source code size and memory usage once you have a model that meets your requirements.

See Chapter 2, “Code Generation and the Build Process,” and Chapter 9, “Optimizing the Model for Code Generation,” to learn techniques for code optimization techniques available for all target configurations. Start by running Model Advisor to evaluate and enhance the quality of code that Real-Time Workshop can generate for a model.

The Real-Time Workshop Embedded Coder documentation contains information about optimization specifically for embedded code.

Make Subsystem and Referenced Model Code Reuseable

If your models contain multiple references to the same atomic subsystem, you can ask Real-Time Workshop to generate a single reentrant function to represent the subsystem, rather than inlining it or generating multiple functions that all do the same thing. Chapter 4, “Building Subsystems and Working with Referenced Models,” tells how to do this, and describes model characteristics that can limit or prevent subsystem reuse. It also tells you how to integrate code from referenced models into your applications and how project (s1prj) directories are organized.

Verify Generated Code

Using Real-Time Workshop data logging features, you can create an executable that runs on your workstation and creates a data file. You can then compare the results of your program with the results of running an equivalent Simulink simulation.

For more information on how to verify Real-Time Workshop generated code, see “Data Import and Export Options” on page 2-13. See also “Tutorial 2: Data Logging” on page 3-19 and “Tutorial 3: Code Verification” on page 3-25 of the Real-Time Workshop Getting Started documentation.

Incorporate Generated Code into Larger Systems

If your Real-Time Workshop generated code is intended to function within an existing code base (for example, if you want to use the generated code as a plug-in function), you should use Real-Time Workshop Embedded Coder. The Real-Time Workshop Embedded Coder documentation describes the entry points and header files you will need to interface your code to Real-Time Workshop Embedded Coder generated code.

Incorporate Existing Code into Generated Code

To interface your hand-written code with Real-Time Workshop generated code, you can use an S-function wrapper. See “Writing S-Functions for Real-Time Workshop” on page 10-1 for specific instructions. For additional details, see the Simulink Writing S-Functions documentation and the Target Language Compiler documentation.

Trace Code Back to Blocks

Real-Time Workshop inserts comments throughout the generated code that make it easy to trace generated code back to your Simulink model. See “Tracing Generated Code Back to Your Simulink Model” on page 2-45 of the Getting Started Guide for more information about this feature. HTML code generation reports and code viewed in Model Explorer include hyperlinks that link symbols in the generated code to blocks that generated them for Real-Time Workshop Embedded Coder users. See “Generate HTML Report” on page 2-5. See “Tutorial 6: Generating Code for a Referenced Model” on page 3-51 for an example of viewing generated code in Model Explorer.

Automate Builds

Using Real-Time Workshop, you can generate code with the push of a button. The automatic build procedure, initiated by a single mouse click and driven by a model and a system target file, generates code, a makefile, and optionally compiles (or cross-compiles) and downloads a program. See “Automatic Program Building” on page 2-13 of the Getting Started documentation for an overview, and Chapter 2, “Code Generation and the Build Process,” for complete details.

You can create your own system target files to create custom targets that interface with external code or operating environments. If you have in the past created system target files, note that the form of callbacks has changed between versions 5 and 6 of Real-Time Workshop. See “System Target Files” in the Real-Time Workshop Embedded Coder targeting documentation for full details.

Tune Parameters During Execution

Parameter tuning enables you to change block parameters while a generated program runs, thus avoiding recompiling the generated code. Real-Time Workshop supports parameter tuning in four different environments:

- **External mode:** You can tune parameters from Simulink while running the generated code on a target processor. See Chapter 6, “External Mode,” for information on this mode.
- **External C application program interface (API):** You can write your own C API interface for parameter tuning using support files provided by The MathWorks. See Chapter 17, “Data Exchange APIs,” for more information.

- **Rapid simulation:** You can use the Rapid Simulation Target (`rsim`) in batch mode to provide fast simulations for performing parametric studies. Although this is not an on-the-fly application of parameter tuning, it is nevertheless a useful way to evaluate a model. This mode is also useful for Monte Carlo simulation. See Chapter 12, “Real-Time Workshop Rapid Simulation Target,” for further information.
- **Simulink:** Prior to generating real-time code, you can tune parameters on the fly in your Simulink model.

See also “Interface with Signals and Parameters” on page 1-18.

Monitor Signals and Log Data

There are several ways to monitor signals and data in Real-Time Workshop:

- **External mode:** You can monitor and log signals from an externally executing program via Scope blocks and several other types of external mode compatible blocks. See “Using the External Mode User Interface” on page 6-3 for a discussion of what external mode can do, and “Creating an External Mode Communication Channel” on page 17-18 for advanced details on customizing external mode communication.
- **External C application program interface (API):** You can write your own C API for signal monitoring using support files provided by The MathWorks. See Chapter 17, “Data Exchange APIs,” for more information.
- **MAT-file logging:** You can use a MAT-file to log data from the generated executable. See “Data Import and Export Options” on page 2-13 for more information.
- **Simulink:** You can use any of the Simulink data logging capabilities.

Interface with Signals and Parameters

You can interface signals and parameters in your model to hand-written code by specifying the storage declarations of signals and parameters. For more information, see

- “Parameters: Storage, Interfacing, and Tuning” on page 5-2
- “Signal Storage, Optimization, and Interfacing” on page 5-19
- “Interfacing Signals to External Code” on page 5-27
- “C-API for Interfacing with Signals and Parameters” on page 17-2

Learn from Sample Implementations

Real-Time Workshop provides sample implementations that illustrate the development of real-time programs under Tornado, as well as generic real-time programs under Windows and UNIX.

These sample implementations are located in the following directories:

- *matlabroot/rtw/c/grt*: Generic real-time examples
- *matlabroot/rtw/c/tornado*: Tornado examples

Code Generation and the Build Process

This chapter provides an overview of the Real-Time Workshop features that you can control with Configuration Set dialogs and Model Explorer. The following sections step you through the Configuration Set dialog panes and discuss further options that control how code is generated, and how it is compiled for specific environments.

The Real-Time Workshop User Interface (p. 2-2)

The features that you control by the **Real-Time Workshop** pane of the **Configuration Parameters** dialog

Simulation Parameters and Code Generation (p. 2-11)

Describes how options on the **Solver**, **Data Import/Export**, **Diagnostics**, and **Real-Time Workshop** panes interact with code generation, and how to trace code back to the blocks that generated it

Selecting a Target Configuration (p. 2-53)

Describes how to use the System Target File Browser, with summaries of target configurations that you can access through the browser

Making an Executable (p. 2-59)

How to control generation of executables during the build process

Choosing and Configuring Your Compiler (p. 2-68)

Aspects of installing a compiler and choosing appropriate template makefiles

Template Makefiles and Make Options (p. 2-71)

Summarizes available template makefiles and make command options

Configuring the Generated Code with TLC (p. 2-77)

Using the Target Language Compiler to generate source code in specific ways or to give the code specific characteristics

The Real-Time Workshop User Interface

Many parameters and options affect the way that Real-Time Workshop generates code from your model and builds an executable. To set these parameters and options, you interact with the panes of the **Configuration Parameters** dialog box.

Parameters set on the Simulink **Solver**, **Data Import/Export**, **Diagnostics**, and **Real-Time Workshop** panes can affect both the behavior of the model in simulation and the code generated from the model. “Simulation Parameters and Code Generation” on page 2-11 discusses how Simulink settings affect the code generation process.

You initiate and control the build process from the **Real-Time Workshop** pane. This pane and its subnodes (also presented as tabs) let you set parameters that directly affect the process of code generation and appearance.

Note When you change a check box, menu selection, or edit field in any **Configuration Parameters** dialog, the white background of the element you altered turns to light yellow to indicate that an unsaved change has been made. When you click **OK**, **Cancel**, or **Apply**, the background is reset to white.

Using the Real-Time Workshop Pane

There are three ways to open the **Real-Time Workshop** pane:

- From the **Simulation** menu, choose **Configuration Parameters** (or type **Ctrl-E**). When the **Configuration Parameters** dialog opens, click **Real-Time Workshop** in the **Select** (left) pane.
- Select **Model Explorer** from the **View** menu in the model window, or type `daexplr` on the MATLAB command line and press **Enter**. In Model Explorer, expand the node for the current model in the left hand pane and click **Configuration (active)**. The configuration dialog elements are listed in the middle pane. Clicking any of these brings up that dialog in the right hand pane. Alternatively, right-clicking the **Real-Time Workshop** configuration element in the middle pane and choosing **Properties** from the context menu activates that dialog in a separate window.

- Select **Options** from the **Real-Time Workshop** submenu of the **Tools** menu in the model window.

The general **Real-Time Workshop** pane is shown below.

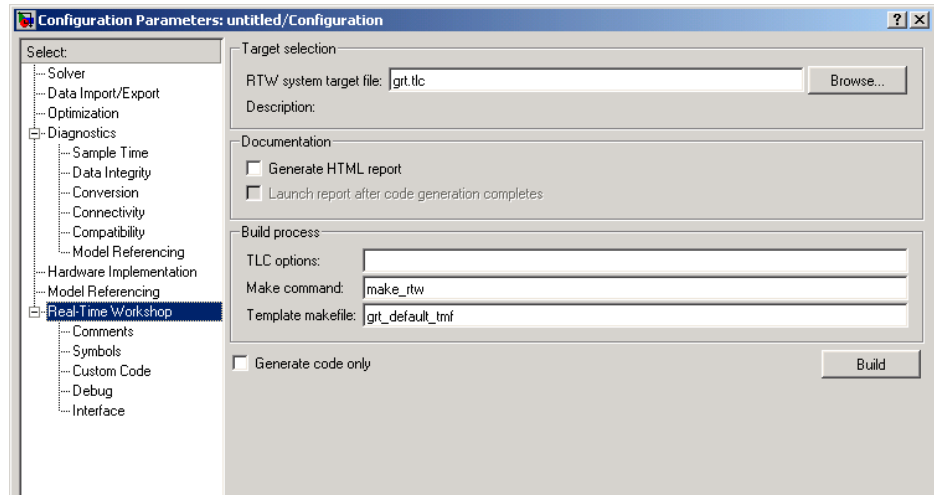


Figure 2-1: Real-Time Workshop Pane

Most **Real-Time Workshop** subpanes are divided into two or three sections. The lowest section, which is present on all subpanes, contains the **Build** (or **Generate Code**) button. For information on the contents of the subpanes, see the following sections:

- “Comments Options” on page 2-37
- “Symbols Options” on page 2-38
- “Custom Code Options” on page 2-39
- “Debug Options” on page 2-40
- “Interface Options” on page 2-42

Build Button

Click the **Build** button to initiate the code generation and build process. The following methods of initiating a build are exactly equivalent to clicking the **Build** button:

- Select **Build Model** from the **Real-Time Workshop** submenu of the **Tools** menu in the Simulink window (or use the key sequence **Ctrl+B**).
- Invoke the `rtwbuild` command from the MATLAB command line. The syntax of the `rtwbuild` command is

```
rtwbuild modelname
```

or

```
rtwbuild('modelname')
```

where `modelname` is the name of the source model. If the source model is not loaded into Simulink, `rtwbuild` loads the model.

Note When **Generate code only** is selected on the **Real-Time Workshop** pane, the **Build** button's name changes to **Generate code**.

Getting Context-Sensitive Help with Tooltips

The general **Real-Time Workshop** pane and its subpanes provide tooltips. Place your pointer over any edit field name or check box to display a message box that briefly explains the option. The message disappears after a few seconds. To see it again, move the pointer slightly.

The following sections summarize each option on the general **Real-Time Workshop** pane.

Target, Documentation, and Build Options

Browse

The **Browse** button opens the System Target File Browser (See Figure 2-14, The System Target File Browser, on page 2-54). The browser lets you select a preset target configuration consisting of a system target file, template makefile, and make command.

“Selecting a Target Configuration” on page 2-53 describes the use of the browser and includes a complete list of available target configurations.

RTW System Target File

The **RTW System target file** field has these functions:

- If you have selected a target configuration using the System Target File Browser, this field displays the name of the chosen system target file (*target.tlc*).
- If you are using a target configuration that does not appear in the System Target File Browser, you must enter the name of the desired system target file in this field.

After the system target filename, you can enter code generation options and variables for the Target Language Compiler. See “Target Language Compiler Variables and Options” on page 2-77 for details.

You can type the name of a system target file into the the **RTW System target file** field directly, and then click **Apply** or **OK** to configure for that target. Note, however, that this method of selection does not add any flags to the **Make command** field, as do some of the targets available in the System Target File Browser.

Generate HTML Report

If you select the **Generate HTML report** option, Real-Time Workshop produces a code generation report in HTML format and automatically opens it for viewing in the MATLAB Help browser. The contents of the report vary from one target to another, but all reports contain the following code generation details:

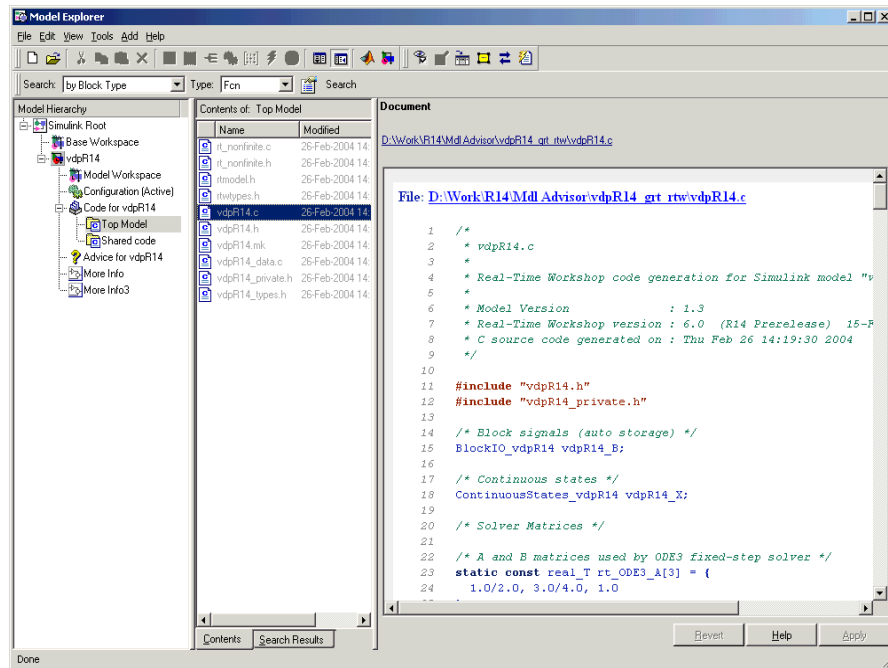
- The Summary section lists version and date information, TLC options used in code generation, and Simulink model settings.
- The Generated Source Files section contains a table of source code files generated from your model. You can view the source code in the MATLAB Help browser. When Real-Time Workshop Embedded Coder is installed, hyperlinks are placed within the source code that let you trace lines of code back to the blocks or subsystems from which the code was generated. Click the hyperlinks to highlight the relevant blocks or subsystems in a Simulink model window.

When you select the **Generate HTML report** check box, Real-Time Workshop automatically selects the check box under it named **Launch report after code generation completes**. When this second check box is selected, Real-Time

Workshop displays the report immediately after it generates the code. If you do not want to see the report at that time, clear this second check box. In either case, you can open the report later. Real-Time Workshop places the report on the MATLAB path at `modname_targetconfigname_rtw\html\modname_codegen_rpt.html`. For example, if the model for which you generated code is named `fuelsys.mdl`, and you selected `grt` as the target configuration, the HTML report will be found in `fuelsys_grt_rtw\html\fuelsys_codegen_rpt.html`.

Viewing Generated Code in Model Explorer. Another way to view the HTML source code report is to use the Code Viewer that is built into Model Explorer. You can browse files generated by Real-Time Workshop, Real-Time Workshop Embedded Coder, and other products 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 directory, the **Hierarchy** (left) Pane of Model Explorer contains a node named `Code` for `model`. Under that node will be other nodes, typically called `Top Model` and `Shared Code`. Clicking `Top Model` displays in the **Contents** (middle) pane a list of source code files in the build directory of each model that is currently open. The figure below shows code for the `vdp` model:



In this example, the file `./vdpR14_grt_rtw/vdpR14.c` is being viewed. To view any 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. A hyperlink for the source file (not the HTML version you are looking at) being viewed sits above it. Clicking it opens that file in a text editing window where you can modify its contents. This is not something you typically do with generated source code, but in the event you have placed custom code files in the build directory, you can edit them as well in this fashion. You can also take advantage of your editor's features such as multipane display or custom syntax coloring.

If an open model contains Model blocks, and if generated code for any of these models exists in the current `s1prj` directory, 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.

The node directly under the Top Model node is named Shared Code. It collects any files that exist in the appropriate `./slprj/target/_sharedutils` subdirectory, containing shared fixed-point utility code.

Note Currently, you cannot use the **Search** tool built into Model Explorer’s toolbar to search generated code displayed in the Code Viewer. However, typing **Ctrl-F** when focused on the **Document** pane pops up a **Find** dialog you can use to search for strings in the currently displayed file.

Getting Model Optimization Advice. The Model Advisor is a tool that helps you configure any model to optimally achieve your code generation objectives. Clicking the icon labeled *Advice* on *model* in the **Model Hierarchy** pane launches the Model Advisor from Model Explorer. This node is directly below the Code for model node, as the above figure shows. Clicking the Advice node causes the **Dialog** pane to be labeled Model Advisor, and to contain a link, **Start model advisor**. When you click that link, Model Advisor (formerly called Model Assistant) opens a separate HTML window with a set of button and check box controls.

Another way to invoke Model Advisor is to type

```
ModelAdvisor('model')
```

specifying the name of an open model, at the MATLAB prompt.

See “Using the Model Advisor” on page 9-4 for further information on Model Advisor.

TLC Options

You can enter Target Language Compiler (TLC) command line options in the **TLC options** edit field, for example

- `-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 (e.g., when debugging)

For additional information, see the Target Language Compiler documentation.

Make Command

A high-level M-file command, invoked when a build is initiated, controls the Real-Time Workshop build process. Each target has an associated make command. The **Make command** field displays this command.

Almost all targets use the default command, `make_rtw`. “Targets Available from the System Target File Browser” on page 2-55 lists the make command associated with each target.

Third-party targets might supply another make command. See the vendor’s documentation.

In addition to the name of the make command, you can supply arguments in the **Make command** field. These arguments include compiler-specific options, include paths, and other parameters. When the build process invokes the make utility, these arguments are passed along in the make command line.

“Template Makefiles and Make Options” on page 2-71 lists the **Make command** arguments you can use with each supported compiler.

Template Makefile

The **Template makefile** field has these functions:

- If you have selected a target configuration using the System Target File Browser, this field displays the name of an M-file that selects an appropriate template makefile for your development environment. For example, in Figure 2-1, the **Template makefile** field displays `grt_default_tmf`, indicating that the build process will invoke `grt_default_tmf.m`. “Template Makefiles and Make Options” on page 2-71 gives a detailed description of the logic by which Real-Time Workshop selects a template makefile.
- Alternatively, you can explicitly enter the name of a specific template makefile (including the extension) in this field. You must do this if you are using a target configuration that does not appear in the System Target File Browser. This is necessary if you have written your own template makefile for a custom target environment.

If you specify your own template makefile, be careful to include the filename extension. If a filename extension is not included in the **Template makefile** field, Real-Time Workshop attempts to find and execute a file with the extension `.m` (i.e., an M-file).

Generate Code Only

When you select this option, the build process generates code but does not invoke the make command. The code is not compiled and an executable is not built.

When you select this option, the caption of the **Build** button changes to **Generate code**.

Simulation Parameters and Code Generation

This section discusses how the simulation parameters of your model interact with Real-Time Workshop code generation. Only simulation parameters that affect code generation are mentioned here. For a full description of simulation parameters, see the Simulink documentation.

This discussion is organized around the panes of the **Configuration Parameters** dialog, and covers the following:

“Solver Options” on page 2-11	Using the Solver pane
“Data Import and Export Options” on page 2-13	Using the Data Import/Export pane
“Optimization Options” on page 2-17	Using the Optimization pane
“Diagnostics Options” on page 2-26	Using the Diagnostics pane
“Hardware Implementation Options” on page 2-28	Using the Hardware Implementation pane
“Model Referencing Options” on page 2-34	Using the Model Referencing pane
“Real-Time Workshop Options” on page 2-35	Using the Real-Time Workshop pane

To view these panes, choose **Configuration Parameters** from the **Simulation** menu or type **Ctrl+E**. When the dialog opens, click the appropriate node in the left-hand pane to work with the dialog in the right-hand pane.

Solver Options

The **Solver** pane of the **Configuration Parameters** dialog is shown below.

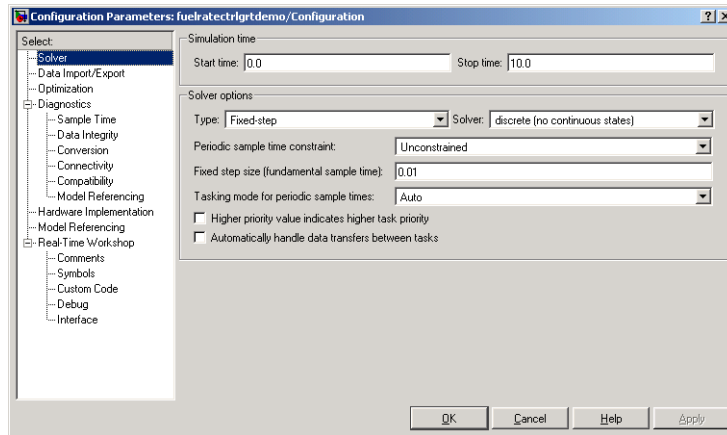


Figure 2-2: Solver Pane for Fixed Step Solver

Start and Stop Times

The stop time must be greater than or equal to the start time. If the stop time is zero, or if the total simulation time (Stop-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.

Note that when using the GRT or Tornado targets, you can override the stop time when running a generated program from the Windows command prompt or UNIX command line. To override the stop time that was set during code generation, use the `-tf` switch.

```
model -tf n
```

The program will run for `n` seconds. If `n = inf`, the program will run indefinitely. See “Part 3: Running the External Mode Target Program” on page 3-45 of the Real-Time Workshop Getting Started documentation for an example of the use of this option.

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 (i.e., it uses `simstructs` instead), you must not use these blocks. See Appendix A, “Blocks That Depend on Absolute Time,” 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 ensure that timers do not overflow and that they use optimal word sizes by specifying the **Application lifespan** parameter on the **Optimization** pane. See “Application Lifespan” on page 2-22 for details.

Type

If you are using an S-function or an RSim target, you can specify either a fixed-step solver or a variable-step solver. All other targets require a fixed-step solver.

Tasking Mode for Periodic Sample Times

Real-Time Workshop supports both single- and multitasking modes. See “Models with Multiple Sample Rates” on page 8-1 for full details.

Data Import and Export Options

This section discusses several different methods by which a Real-Time Workshop generated program can save data to a MAT-file for later analysis. These methods include

- Using the Data Import/Export pane to define and log workspace return variables
- Logging data from Scope and To Workspace blocks
- Logging data using To File blocks

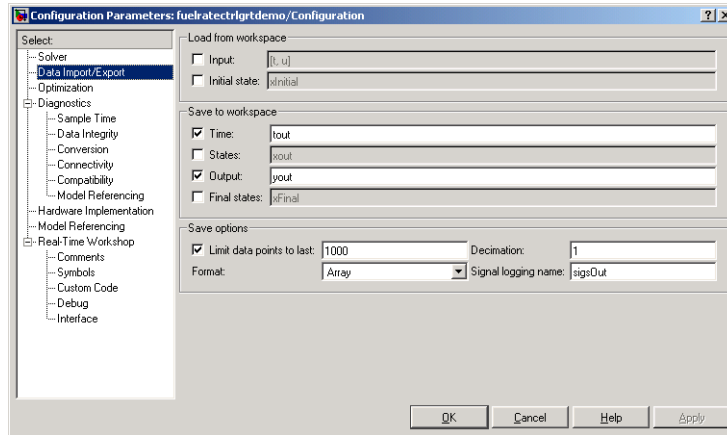


Figure 2-3: The Data Import/Export Pane

“Tutorial 2: Data Logging” on page 3-19 of the Real-Time Workshop Getting Started documentation is an exercise designed to give you hands-on experience with data logging features of Real-Time Workshop.

Note Data logging is available only for targets that have access to a file system. In addition, only the RSim target executables are capable of accessing MATLAB workspace data; GRT and ERT targets cannot.

Logging States, Time, and Outputs via the Data Import/Export Pane

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 the Simulink documentation.

For each workspace return variable that you define and enable, Real-Time Workshop 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`.

Real-Time Workshop logs the following data:

- All root Output 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 Output block, starting with 1.
- All 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`.

Real-Time Workshop data logging follows the **Data Import/Export** Save options (**Limit data points to last**, **Decimation**, **Format**, and **Signal Logging Name**).

Overriding Default MAT-File Name. The MAT-file name defaults to `model.mat`. To specify a different filename,

- 1 Choose **Configuration Parameters** from the **Simulation** menu. The dialog box opens. Click **Real-Time Workshop**.
- 2 Append the following option to the existing text in the **Make command** field.

```
OPTS=" -DSAVEFILE=filename"
```

Overriding Default MAT-File Variable Names. By default, Real-Time Workshop prefixes the string `rt_` to the variable names for system outputs, states, and simulation time to form MAT-file variable names. To change this prefix,

- 1 Choose **Configuration Parameters** from the **Simulation** menu. The dialog box opens. Click **Real-Time Workshop**.
- 2 In the **RTW system target** field, select `grt.tlc`.
- 3 Under **Real-Time Workshop**, select the **Interface** subpane.

- 4 Select a prefix (`rt_`) or suffix (`_rt`) from the **MAT-file variable name modifier** field, or choose none for no prefix (note that other targets may or may not have this option).

Logging Data with Scope and To Workspace Blocks

Real-Time Workshop also logs data from these sources:

- All Scope blocks that have the **save data to workspace** option enabled
You must specify the variable name and data format in each Scope block's dialog box.
- All 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 any variables logged from the Workspace I/O pane.

Logging 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's time and input variable(s). You must specify the filename, variable names, decimation, and sample time in the To File block's dialog box.

Note Models referenced by Model blocks do not perform data logging in that context except for states, which you can include in state logged for top models. RTW generated code for referenced models thus does not perform data logging to MAT-files.

Data Logging Differences in Single- and Multitasking Models

When logging data in single-tasking and multitasking systems, you will notice differences in the logging of

- Noncontinuous root Output 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, Real-Time Workshop logs states and outputs after the first time step.

See “Data Logging in Single Tasking and Multitasking Model Execution” on page 7–14 for more details on the differences between single- and multitasking data logging.

Note The rapid simulation target (RSim) provides enhanced logging options. See Chapter 12, “Real-Time Workshop Rapid Simulation Target” for more information.

Optimization Options

The figure below shows the **Optimization** pane. This pane includes several options that affect the performance of generated code. These help you to optimize memory usage, code size, and efficiency.

When you change targets, other options might appear lower down on the **Optimization** dialog. For examples of ERT options, see the Real-Time Workshop Embedded Coder release notes.

The following subsections discuss the **Optimization** options. For additional details and examples, refer to “Optimizing the Model for Code Generation” on page 9-1.

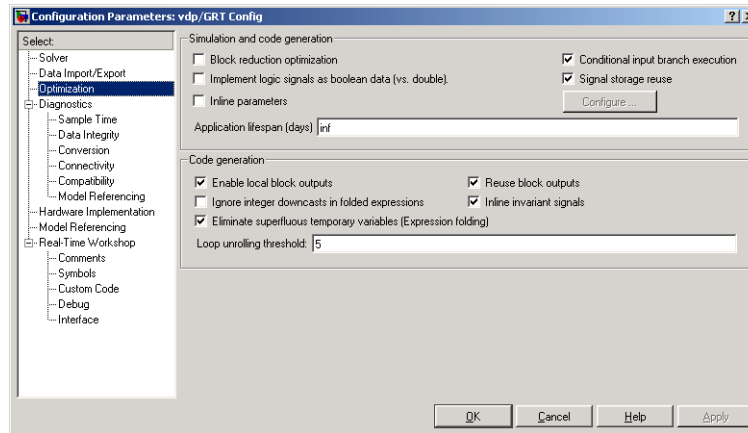


Figure 2-4: Optimization Pane

Block Reduction Optimization

When this check box is selected, Simulink collapses certain groups of blocks into a single, more efficient block, or removes them entirely. This results in faster model execution during simulation and in generated code. The appearance of the source model does not change.

By default, the **Block reduction optimization** check box is selected.

The types of block reduction optimizations currently supported are

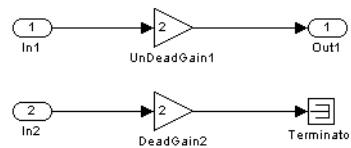
Accumulator Folding. Simulink recognizes certain constructs such as accumulators, and reduces them to a single block. For a detailed example, see “Accumulators” on page 9-28.

Removal of Redundant Type Conversions. Unnecessary type conversion blocks are removed. For example, an `int` type conversion block whose input and output are of type `int` is redundant and will be removed.

Dead Code Elimination. Any blocks or signals in an *unused code path* are eliminated from the generated code if the **Block reduction optimization** check box is selected. There are three conditions that all need to be met for a block to be considered part of an unused code path:

- 1 The block is in a signal path that ends with a Terminator block or a disabled Assertion block.
- 2 The block is not in any other signal path.
- 3 The block does not reference any tunable or global parameters or signal storage.

Consider the model in the following block diagram.



Code is always generated for the signal path between In1 and Out1, because this path does not meet condition 1 above. If **Inline parameters** is cleared, code is also generated for the signal path between the In2 and Terminator blocks, because condition 3 is not satisfied (Gain2 is tunable).

If **Inline parameters** is selected, however, the terminated signal path meets all three conditions, and is eliminated. Code for the resultant function, `dead_gain_output`, (which, as shown, is called by the GRT wrapper function `MdlOutputs`) is presented below:

```

/* wrapper function */
void MdlOutputs(int_T tid) {

    dead_gain_output(tid);
}

/* model output */
static void dead_gain_output(int_T tid)
{

    /* local block i/o variables */

    real_T rtb_UnDeadGain1_x;

```

```
/* Gain: '<Root>/UnDeadGain1' */
rtb_UnDeadGain1_x = dead_gain_U.In1 * 2.0;

/* Outport: '<Root>/Out1' */
dead_gain_Y.Out1 = rtb_UnDeadGain1_x;
}
```

Conditional Input Branch Execution

This optimization factors out unneeded code that is upstream from Switch blocks. When Conditional input branch optimization is on, instead of executing all blocks driving the Switch block input ports at each time step, only the blocks required to compute the control input and the data input selected by the control input are executed.

You control conditional input branch optimization by selecting and deselecting the **Conditional input branch execution** check box on the **Optimization** pane of the **Configuration Parameters** dialog box.

For further information on using this optimization, see “Conditional Input Execution” on page 9-16.

Implement Logic Signals as Boolean Data

By default, Simulink does not signal an error when it detects that double signals are connected to blocks that prefer Boolean input. This ensures compatibility with models created by earlier versions of Simulink that support only double data types. You can enable strict Boolean type checking by selecting the **Implement logic signals as boolean data (vs. double)** check box.

Selecting this check box is recommended. Generated code will require less memory, because a Boolean signal typically requires one byte of storage while a double signal requires eight bytes of storage.

Inline Parameters

Selecting this option has two effects:

- Real-Time Workshop uses the numerical values of model parameters, instead of their symbolic names, in generated code.

If the value of a parameter is a workspace variable, or an expression including one or more workspace variables, the variable or expression is evaluated at code generation time. The hard-coded result value appears in the generated code. An inlined parameter, since it has in effect been transformed into a constant, is no longer tunable. That is, it is not visible to externally written code, and its value cannot be changed at run-time.

- The **Configure** button becomes enabled. Clicking the **Configure** button opens the **Model Parameter Configuration** dialog box.

The **Model Parameter Configuration** dialog box lets you remove individual parameters from inlining and declare them to be tunable variables (or global constants). When you declare a parameter tunable, Real-Time Workshop generates a storage declaration that allows the parameter to be interfaced to externally written code. This enables your hand-written code to change the value of the parameter at run-time.

The **Model Parameter Configuration** dialog box lets you improve overall efficiency by inlining most parameters, while at the same time retaining the flexibility of run-time tuning for selected parameters.

See “Parameters: Storage, Interfacing, and Tuning” on page 5-2 for further information on interfacing parameters to externally written code.

The **Inline parameters** check box also instructs Simulink to propagate constant sample times. Simulink computes the output signals of blocks that have constant sample times once during model startup. This improves performance, since such blocks do not compute their outputs at every time step of the model.

Note **Inline parameters** must be *on* for referenced models. You cannot use the **Model Parameter Configuration** dialog box to tune parameters that are passed to referenced models. To tune such parameters, you must declare them in the referenced model’s workspace, and then pass run-time values (or expressions) for them in argument lists specified for each Model block that references that model. See “Using Model Arguments” in the Simulink documentation for specific details.

You can select the **Inline invariant signals** code generation option (which also places constant values in generated code) only when **Inline parameters** is *on*. See “Inline Invariant Signals” on page 2-23.

Signal Storage Reuse

This option instructs Real-Time Workshop to reuse signal memory. This reduces the memory requirements of your real-time program. We recommend selecting this option. Disabling **Signal storage reuse** makes all block outputs global and unique, which in many cases significantly increases RAM and ROM usage.

For further details on the **Signal storage reuse** option, see “Signal Storage, Optimization, and Interfacing” on page 5-19.

Note Selecting **Signal storage reuse** also enables the **Enable local block outputs** option and the **Reuse block outputs** option in the Code generation Signals category of the **Optimization** pane. See “Enable Local Block Outputs” on page 2-23 and “Reuse Block Outputs” on page 2-23.

Application Lifespan

The **Application lifespan (days)** field lets you specify how long an application that contains blocks that depend on elapsed time should be able to execute before timer overflow. Specifying a lifespan determines the word size used by timers in the generated code, and can lower RAM usage.

Application lifespan, when combined with the step size of each task, determines the data type used for integer absolute time for each task, as follows:

- If your model does not require absolute time, this option affects neither simulation nor the generated code.
- If your model requires absolute time, this option optimizes the word size used for storing integer absolute time in generated code. This ensures that timers will not overflow within the lifespan you specify. If you set **Application lifespan** to *Inf*, two `uint32` words are used.
- If your model contains fixed-point blocks that require absolute time, this option affects both simulation and generated code.

Using 64 bits to store timing data enables models with a step size of 0.001 microsecond (10E-09 seconds) to run for more than 500 years, which would rarely be required. To run a model with a step size of one millisecond (0.001 seconds) for one day would require a 32-bit timer (but it could continue running for 49 days). **Application lifespan** was an ERT-only option in prior releases.

Enable Local Block Outputs

When this option is selected, block signals are declared locally in functions instead of being declared globally (when possible).

Note This check box is unavailable when the **Signal storage reuse** check box is cleared.

For further information on the use of the **Enable local block outputs** option, see “Signal Storage, Optimization, and Interfacing” on page 5-19. Also go through “Tutorial 4: A First Look at Generated Code” on page 3-30 of the Getting Started documentation if you have not done so already.

Reuse Block Outputs

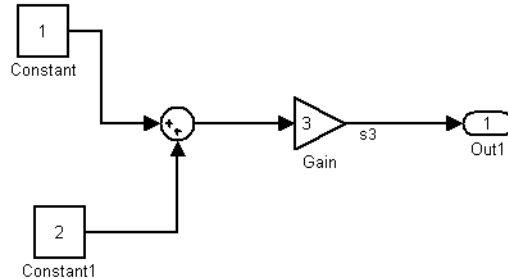
When the **Reuse block output** check box is selected (the default) Real-Time Workshop reuses signal memory whenever possible. When **Reuse block output** is cleared, signals are stored in unique locations.

Note that **Reuse block output** is available only when the **Signal storage reuse** check box is selected.

See “Signal Storage, Optimization, and Interfacing” on page 5-19 for further information (including generated code example) on **Reuse block output** and other signal storage options.

Inline Invariant Signals

An invariant signal is a block output signal that does not change during Simulink simulation. For example, the signal S3 in this block diagram is an invariant signal.



For the model above, if **Inline invariant signals** is selected on the **Optimization** pane, Real-Time Workshop inlines the invariant signal S3 in the generated code.

Note that an *invariant signal* is not the same as an *invariant constant*. (See the Using Simulink documentation for information on invariant constants.) In the preceding example, the two constants (1 and 2) and the gain value of 3 are invariant constants. To inline these invariant constants, select **Inline parameters**.

The **Ignore integer downcasts in folded expressions** check box formerly was labeled **Enforce integer downcasts**. The default behavior is the same (to perform downcasts in expressions), but the sense of the option's wording and thus its initial state has been changed.

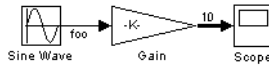
Note If your model contains Model blocks, **Inline parameters** must be on for it and all models it references. If a top model or a referenced model does not have **Inline Parameters** on, Simulink temporarily enables this option while generating code, then turns it off again when the build completes. That is, the model is left in its previous state and need not be resaved.

Loop Unrolling Threshold

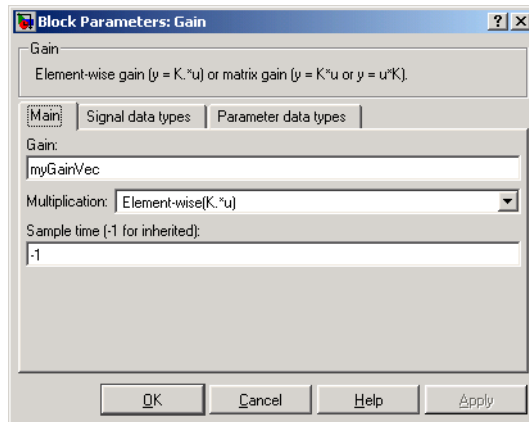
The **Loop unrolling threshold** field on the **Optimization** pane determines when a wide signal or parameter should be wrapped into a for loop and when

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 (identified as `rtP` prior to Version 6). 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 fragment:

```
{
    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 fragment 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 the Target Language Compiler Reference Guide documentation for more information on loop rolling.

Note When a model includes Stateflow charts or Embedded MATLAB Function blocks, a set of Stateflow optimizations appears on the **Optimization** pane. The settings you make for the Stateflow options also apply to all Embedded MATLAB Function blocks in the model. This is because the Embedded MATLAB Function blocks and Stateflow are built on top of the same technology and share a code base. Note that you do not need a Stateflow license in order to use Embedded MATLAB Function blocks.

Diagnosics Options

The figure below shows the main **Diagnosics** pane. This pane specifies what action should be taken when various model conditions, such as unconnected ports, are encountered. You can specify whether to ignore a given condition, issue a warning, or raise an error. If an error condition is encountered during a build, the build is terminated. The section below discusses how you can control Assertion block behavior with this pane. For a full description of the **Diagnosics** pane, see the Simulink documentation.

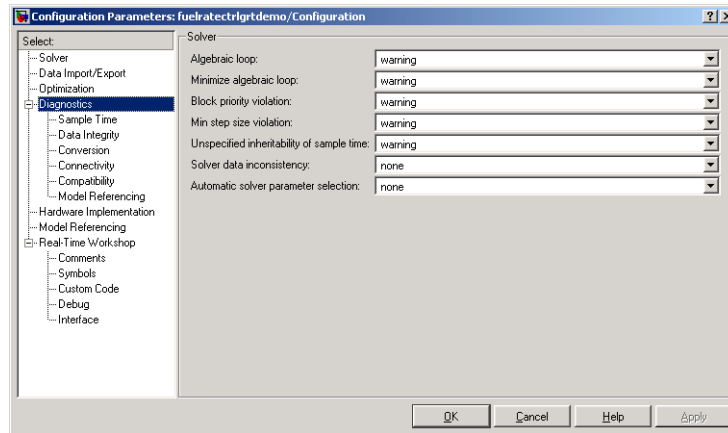


Figure 2-5: Diagnostics Pane

Control over Assertion Block Behavior

The **Model Verification block enabling** menu in the **Data Integrity** subpane specifies whether model verification blocks such as Assert, Check Static Gap, and related range check blocks are included, excluded, or default to their local settings. The diagnostic has the same effect on code generated by Real-Time Workshop as it does on simulation behavior. Settings are

- Use local settings
- Enable all
- Disable all

For Assertion blocks that are not disabled, the generated code for a model will include one of the following statements,

```
utAssert(input_signal);
utAssert(input_signal != 0.0);
utAssert(input_signal != 0);
```

at appropriate locations, depending on the block's input signal type (Boolean, real, or integer, respectively).

By default `utAssert` is a noop in generated code. For assertions to abort execution you must enable them by including a parameter in the `make_rtw`

command. Specify the **Make command** field on the **Real-Time Workshop** pane as follows:

```
make_rtw OPTS=' -DDOASSERTS'
```

If you want triggered assertions not to abort execution and instead to print out the assertion statement, use the following `make_rtw` variant:

```
make_rtw OPTS=' -DDOASSERTS -DPRINT_ASSERTS'
```

`utAssert` is defined as

```
#define utAssert(exp)  assert(exp)
```

You can provide your own definition of `utAssert` in a hand-coded header file if you want to customize assertion behavior in generated code. See `<matlabroot>/rtw/c/libsrc/rtlibsrc.h` for implementation details.

Finally, when running a model in accelerator mode, Simulink calls back to itself to execute assertion blocks instead of using generated code. Thus, user-defined callbacks are still called when assertions fail.

Hardware Implementation Options

These options allow you to specify the constraints of the target microprocessor, such as word size for C-language integer types and byte ordering. You can set them to be the same or different for simulation and code generation. Real-Time Workshop provides a menu of specific microprocessors you can choose from, or you can specify microprocessor characteristics directly, in which case you identify the bit size of each C data type and other characteristics of that target. The **Hardware Implementation** pane is shown below:

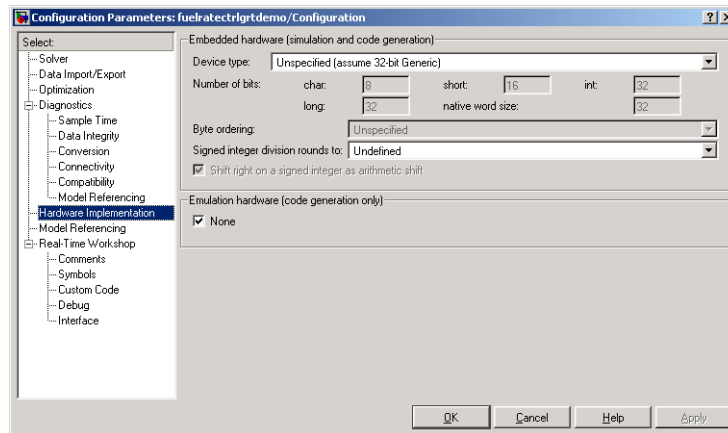


Figure 2-6: Hardware Implementation Pane

Code generation targets can have different word sizes and other hardware characteristics from the MATLAB host. Furthermore, code can be prototyped on hardware that is different from either the deployment target or the MATLAB host. Real-Time Workshop is able to take these differences into account when generating code. Previously, this was accomplished by placing “hook files” on the MATLAB path that specify target hardware characteristics, as described in the Real-Time Workshop 5.0 documentation. Hook files are being deprecated in Release 14. The information they contain is subsumed in the **Hardware Implementation Configuration Parameters** dialog. See “Hook File Compatibility” on page 2-33 for a description of how Version 6 handles existing hook files.

Using the **Hardware Implementation** configuration component, you simultaneously specify integer and fixed-point numerical behavior for two devices:

- **Embedded Hardware (simulation and code generation)** — The deployment hardware device for the model and the RTW generated code. Specifying this information in Simulink allows it to properly simulate the behavior the user can expect on eventual hardware device.
- **Emulation Hardware (Code Generation Only)** — The device on which code generated by Real-Time Workshop currently runs. Rapid prototyping can be done on hardware devices that do not match the final hardware device

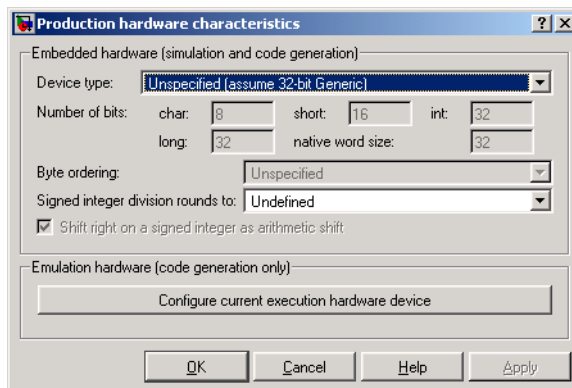
characteristics. The code generation process uses the prototyping hardware device characteristics in conjunction with the deployment hardware device characteristics to generate code that will behave just like it will on the deployment device.

The latter uses the former's specification by default.

For additional information on the **Hardware** pane, run the `rtwdemo_targetsettings` demo.

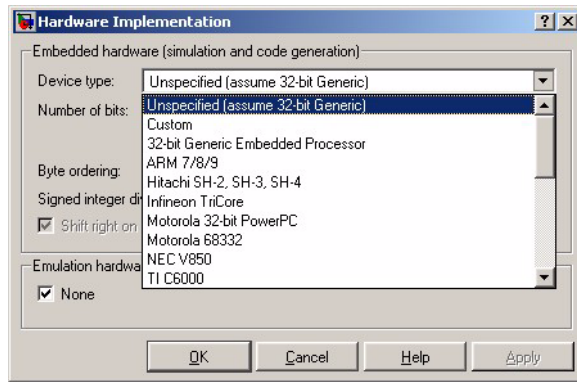
Hardware Configuration for Pre-Version 6 Models

When you open a preexisting model that has not been saved using the current version of Simulink, and select the **Hardware Configuration Parameters** dialog, the following set of controls appears:



Default Configuration of the Hardware Pane for a Preexisting Model

Note that all but one of the parameters below the **Device type** menu are grayed out. This is because these characteristics have been preset for the default target (32-bit Generic), as well as for several dozen known target processors that you can select from that menu, as illustrated below:



Selecting a Final Embedded Hardware Device

In the event that none of the choices listed in the **Device Type** drop-down menu is appropriate for your intended hardware target, you can select **Custom**, and then set values for the hardware characteristics. Selecting any other option disables them. The hardware characteristics that you can specify are

- **Number of bits** — Text fields that specify the number of bits used to represent C types **char**, **short**, **int**, and **long**.
- **Byte ordering** — Specifies whether the target hardware uses **Big Endian** (most significant byte first) or **Little Endian** (least significant byte first) byte ordering. If left as **Unspecified**, Real-Time Workshop generates code to determine the endianness of the target; this is the least efficient option.
- **Shift right on a signed integer as arithmetic shift** — ANSI C leaves the behavior of right shifts on negative integers as implementation dependent. Use this control to specify how Real-Time Workshop implements right shifts on signed integers in generated code.

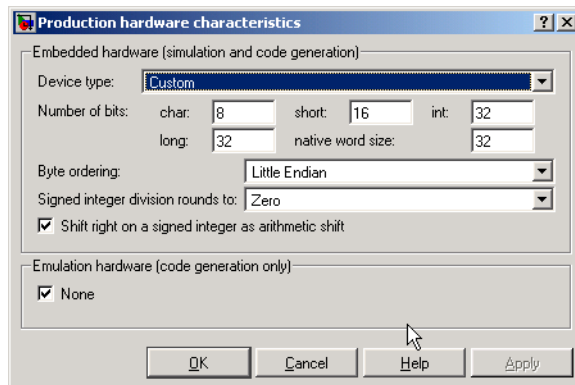
The option is selected by default. If your C compiler handles right shifts as arithmetic shifts, this is the preferred setting.

- When the option is selected, Real Time Workshop generates simple efficient code whenever the Simulink model performs arithmetic shifts on signed integers.
- When the option is unselected, Real Time Workshop generates fully portable but less efficient code to implement right arithmetic shifts.

Emulation Hardware (Code Generation Only)

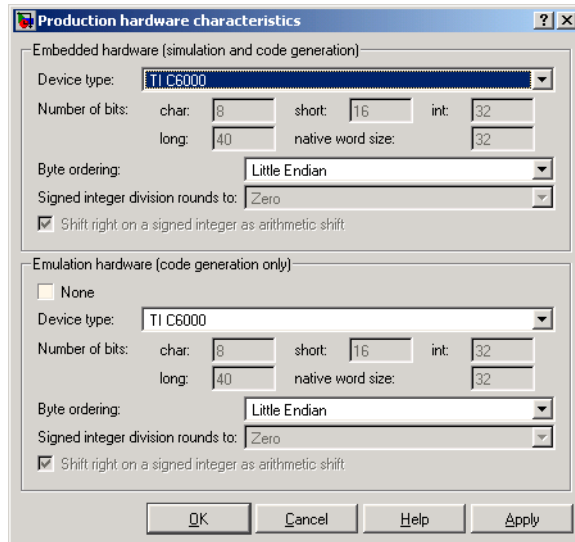
This second group of **Hardware Implementation** pane controls governs how hardware characteristics are handled in generated code. They do not appear unless Real-Time Workshop is installed. Their appearance varies depending on whether hardware configuration characteristics were previously specified for the model or not. If they were not, you will see a button (as illustrated in the first of the two above figures) labeled **Configure current execution hardware device**. Note that this button never again appears for this model once code has been generated and the model has been saved.

When you click the **Configure current execution hardware device** button, it is replaced by a check box labeled **None**. This box is selected by default, as shown in the following figure. For details on what this button does, see “Hook File Compatibility” on page 2-33.



Emulation Hardware Device Defaulted to the Embedded Hardware Device

If you deselect this box, controls appear for that section that are identical to the controls for the **Embedded Hardware** section above, as shown below (in this example the TI -C6000 processor is selected).



Selecting Emulation Hardware Characteristics

Hook File Compatibility

Real-Time Workshop now only reads existing hook files when a model created by Version 5 (Release 13) is built for the first time in Version 6 without first specifying characteristics of the **Current code generation execution hardware device** (see description above). If you build a model in this underspecified state, Real-Time Workshop scans the current directory, then the MATLAB path, for an existing hook file with the name `<target>_rtw_info_hook.m`. If the file is found, its instructions override the defaults in that section. You can subsequently respecify any characteristic freely. If at any point prior to building the target code you do specify **Current code generation execution hardware device**, hook files are ignored, as hardware characteristics are now configured.

When a preexisting (pre-Version 6) model is opened, the **Hardware** configuration dialog displays a **Configure current execution hardware device** button. This button disappears after being pressed once. When code is generated (**Ctrl-B**) for the target the model specifies,

- If the target has a hook file, and the **Configure current execution hardware device** button has not yet been pressed,
 - The hook file is executed and configures the fields specifying current code generation execution hardware device.
 - A warning is issued to the user that the hook file was used.
 - The **Configure current execution hardware device** button on the **Hardware** configuration dialog is permanently removed for that model (assuming that you save the model).
- If the target has a hook file and the **Configure current execution hardware device** button has been pressed (removing it),
 - Code is generated for the target using the hardware characteristics for the current code generation execution hardware device (which can default to those of the final embedded hardware device).
 - The hook file for the target is ignored, and is from now on;
 - A warning is issued that a hook file exists but was not used;
- If the target has no hook file, no message to that effect is issued, and the current code generation execution hardware device, if left unspecified, is mirror the embedded hardware device (which defaults to 32-bit Generic Embedded Processor).

Model Referencing Options

Simulink allows you to include models in other models as blocks, a feature called model referencing. The **Model Referencing** pane allows you to specify options for including other models in this model and this model in other models, and for building simulation and code generation targets. For details see the Simulink documentation.

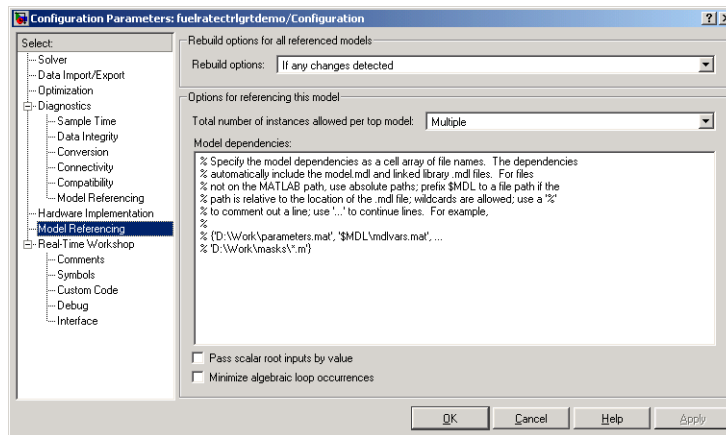


Figure 2-7: Model Referencing Pane

For information on the **Minimize algebraic loop occurrences** option, see the discussion of “Algebraic Loops” on page 2-50.

Real-Time Workshop Options

The figure below shows the top-level **Real-Time Workshop** pane from the **Configuration Parameters** dialog.

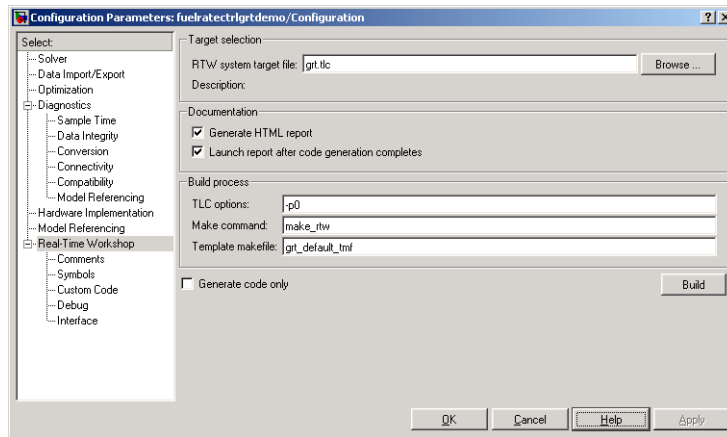


Figure 2-8: General Real-Time Workshop Pane

This pane allows you to specify most of the options for Real-Time Workshop to generate files as desired. Notice the subpanes listed on the left pane of this dialog box under **Real-Time Workshop**. The subpane names are

- **Comments**
- **Symbols**
- **Custom code**
- **Debug**
- **Interface**

The contents of a pane can change when you choose a different target. Thus, a model containing several configuration sets can invoke parameters in one configuration that do not apply to another configuration. In addition, some configuration options are available only with Real-Time Workshop Embedded Coder.

The following figures use stand-alone views of configuration options displayed as a tabbed dialog box. To obtain this dialog box, open Model Explorer, select a configuration set in the **Hierarchy** pane, then right-click the **Real-Time Workshop** node in the **Contents** pane, and select **Properties** from the pop-up menu.

Comments Options

The figure below shows the **Comments** pane, which controls whether and how comments are generated into code.

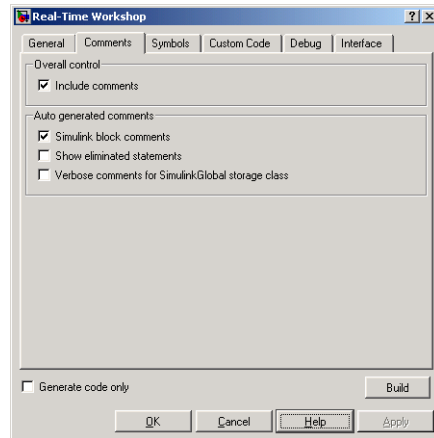


Figure 2-9: Comments Pane

Include Comments. This check box determines whether any comments are placed in the generated files. Selecting this check box allows you to select one or more of the comment types indicated in the **Auto-generated comments** pane to be placed in the generated code. If you clear this check box, the comments do not appear in the generated files.

Simulink Block Comments. When selected, this check box allows the automatically generated comments that describe a block's code to precede that code in the generated file.

Show Eliminated Statements. If this option is selected, statements that were eliminated as the result of optimizations (such as parameter inlining) appear as comments in the generated code. The default is not to include eliminated statements.

Verbose Comments for SimulinkGlobal Storage Class. This check box controls the generation of comments in the model parameter structure declaration in *model_prm.h*. Parameter comments indicate parameter variable names and the names of source blocks. When this check box is cleared (the default),

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. When this check box is selected, parameter comments are generated regardless of the number of parameters.

Symbols Options

The figure below shows the **Symbols** pane, which you use to control how identifiers and objects are named.

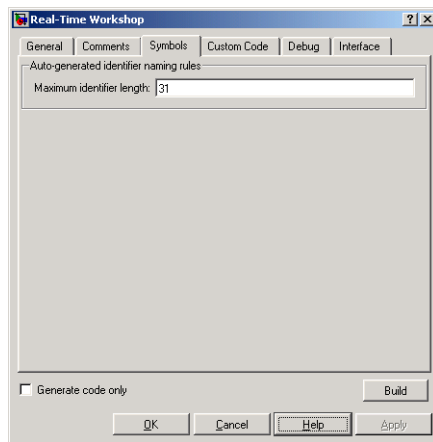


Figure 2-10: Symbols Pane

Maximum Identifier Length. This is the only **Symbols** option for GRT targets. The **Maximum identifier length** field allows you to limit the number of characters in function, typedef, and variable names. The default is 31 characters. This is also the minimum length you can specify; the maximum is 256 characters. Consider increasing identifier length for models having a deep hierarchical structure, as well as when exercising some of the mnemonic identifier options described below.

Within a model containing Model blocks, there can be no collisions between the names of the constituent models. When generating code from a model that uses model referencing, the **Maximum identifier length** must be large enough to accommodate the root model name and the name mangling string (if any). 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.

Custom Code Options

The figure below shows the **Custom Code** pane, which you can use to include your own headers, files, and functions in generated code.

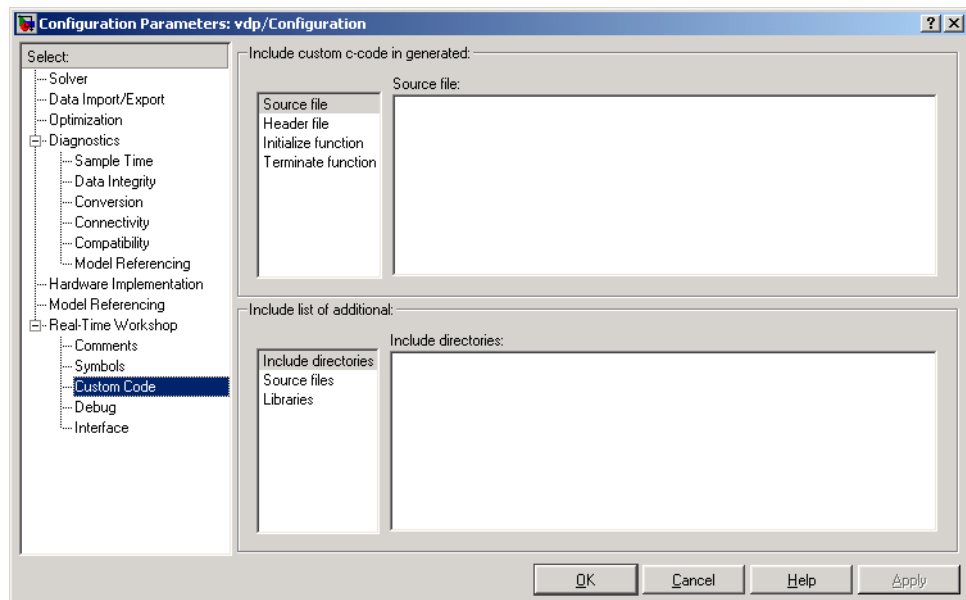


Figure 2-11: Custom Code Pane

Use the **Custom Code** dialog to insert code fragments into the generated files and to include additional files and paths in the build process. The sections and subsections of the dialog are

- Custom C-code inserted into the specified file or function
 - **Source file**
Code is placed near the top of the generated *model.c* file, outside of any function.

- **Header file**
Code is placed near the top of the generated *model.h* file.
- **Initialize function**
Code is placed inside the model's initialize function in the *model.c* file.
- **Terminate function**
Code is placed inside the model's terminate function in the *model.c* file.
Note that you should also select the **Terminate function required** check box on the **Real-Time Workshop Interface** pane.
- Additional files and paths to be added into the build process
 - **Include directories**
List of additional include directories where header files can be found.
Specify full paths to the directory.
 - **Source files**
List of additional source files to be compiled and linked with. The files can be specified with a full path, although a filename alone suffices if the file is located in the current MATLAB directory or in one of the Include directories.
 - **Libraries**
List of additional libraries to be linked with. The libraries can be specified with a full path or just filename when located in the current MATLAB directory or is listed as one of the Include directories.

Note Custom code that you include in a configuration set is ignored when building Accelerator, S-Function, and Model Reference Simulation Targets.

Debug Options

The **Debug** tab controls options to help in troubleshooting generated code. It contains two sections:

- RTW Process — The model compilation phase
- TLC Process — The target language code generation phase

The Debug options are of interest to those who are writing TLC code when customizing targets, integrating legacy code, or developing new blocks. These

options are summarized here. Refer to the Target Language Compiler documentation for details. The figure below shows the **Debug** pane.

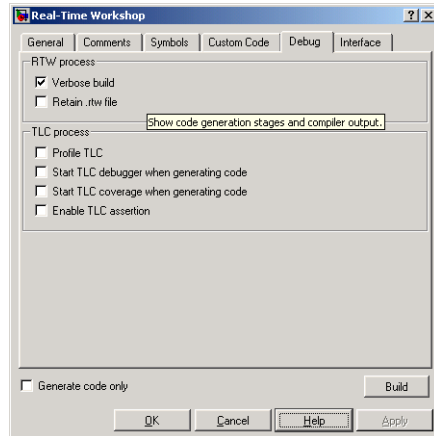


Figure 2-12: Debug Pane, with tool tip

Verbose Build. If this option is selected, the MATLAB Command Window displays progress information during code generation. Compiler output also is made visible.

Retain .rtw File. Normally, the build process deletes the *model*.rtw file from the build directory at the end of the build. When **Retain .rtw file** is selected, *model*.rtw is not deleted. This option is useful if you are modifying the target files, in which case you will need to look at the *model*.rtw file.

You can use this option to generate code with the Target Language Compiler without having to rebuild the target. If you are developing custom TLC scripts for code generation, you can now decrease development time by speeding the edit-generate-inspect cycle when generating code for models that are not changing between iterations. You can bypass rebuilding the model (**Ctrl+B**) if all you are doing is editing TLC files used in the latter part of the code generation process.

To use this new feature, select the **Retain .rtw file** option under the **Real-Time Workshop/Debug** tab in the Model Explorer. The next time you build, the *model*.rtw file is saved in your build directory, along with two other files,

`runtlccmd.m` and `tlccmd.mat`. From that point on, you can invoke the Target Language Compiler independently and with the proper parameters by executing `runtlccmd.m`. The MAT-file is used to store the parameters used by the M-file in issuing the TLC command. You can rebuild the model as required, and this time-saving option remains available as long as you continue to retain your `model.rtw` file each time you build.

Profile TLC. When this option is selected, the TLC profiler analyzes the performance of TLC code executed during code generation, and generates a report. The report is in HTML format and can be read in your Web browser.

Start TLC Debugger when Generating Code. This option starts the TLC debugger during code generation. You can also invoke the TLC debugger by entering the `-dc` argument into the **System Target File** field on the **Real-Time Workshop** pane.

To invoke the debugger and run a debugger script, enter `-df filename` into the **System Target File** field on the **Real-Time Workshop** pane.

Start TLC Coverage when Generating Code. When this option is selected, the Target Language Compiler generates a report containing statistics indicating how many times each line of TLC code is hit during code generation.

This option is equivalent to entering the `-dg` argument into the **System Target File** field on the **Real-Time Workshop** pane.

Enable TLC Assertion. When this box is selected, Real-Time Workshop halts building if any user-supplied TLC file contains an `%assert` directive that evaluates to `FALSE`. The box is not selected by default, meaning that TLC assertion code is ignored. You can also use these MATLAB commands to control TLC assertion handling:

```
set_param(model, 'TLCAssertion', 'on|off') to set this flag on or off.  
Default is off.
```

```
get_param(model, 'TLCAssertion') to see the current setting.
```

Interface Options

The figure below shows the **Interface** pane, which gives you control over which math library is used at run-time, whether to include one of three APIs in generated code, and certain other options.

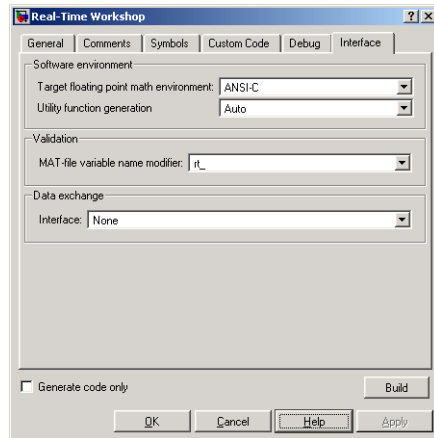


Figure 2-13: Interface Pane

Target Floating Point Math Environment. Target configurations can expressly specify the floating-point math library to use when generating code. Real-Time Workshop uses a switchyard called the Target Function Library (TFL) to designate compiler-specific versions of math functions. The mappings created in the TFL allow for C run-time library support specific to a compiler.

Real-Time Workshop provides three different TFLs:

- `ansi_tf1_tmw.mat` — ANSI-C library (default)
- `iso_tf1_tmw.mat` — Extensions for ISO-C/C99
- `gnu_tf1_tmw.mat` — Extensions for GNU

You choose among them by setting the **Target floating point math environment** pull-down in the **Software Environment** section of the **Interface** tab of the **Real-Time Workshop** configuration set dialog. This enables you to specify different run-time libraries for different configuration sets within a given model.

Selecting ANSI-C provides the ANSI-C set of library functions. For example, selecting ANSI-C would result in generated code that calls `sin()` whether the input argument is double precision or single precision. However, if ISO-C is selected, the call would instead be to the function `sinf()`, which is single

precision. If your compiler supports the ISO-C math extensions, selecting the ISO-C library can result in more efficient code.

Utility Function Generation. Use this drop-down menu to direct where Real-Time Workshop should place fixed-point and other utility code. The choices are `Auto` and `Shared location`. The `shared location` directs code for utilities to be placed within the `slprj` directory in your working directory, which is used for building model reference targets. The `Auto` option operates as follows:

- When the model contains Model blocks, place utilities within the `slprj/target/_sharedutils` directory.
- When the model does not contain Model blocks, place utilities in the build directory (generally, in `model.c`).

MAT-File Variable Name Modifier. This field allows you to select a string to be added to the variable names used when logging data to MAT-files, to distinguish logging data from Real-Time Workshop applications and Simulink. You can select a prefix (`rt_`), suffix (`_rt`), or choose to have no modifier. Real-Time Workshop prefixes or appends the string chosen to the variable names for system outputs, states, and simulation time specified in the **Data Import/Export** pane.

See “Data Import and Export Options” on page 2-13 for information on MAT-file data logging.

Interface. Use this drop-down menu to tell Real-Time Workshop to include one of three APIs when it generates code:

- C-API
- External Mode
- ASAP2

When you select `C-API` or `External Mode`, other options appear. Note that code for only one of these can be generated in a given build. For additional information on working with these interfaces, see “C-API for Interfacing with Signals and Parameters” on page 17-2, “External Mode” on page 6-1, and “Generating ASAP2 Files” on page B-1.

Tracing Generated Code Back to Your Simulink Model

Real-Time Workshop writes system/block identification tags in the generated code. The tags are designed to help you identify the block in your source model that generated a given line of code. Tags are located in comment lines above each line of generated code, and are provided with hyperlinks in HTML code generation reports that you can optionally generate.

The tag format is `<system>/block_name`, where

- `system` is either
 - The string 'root', or
 - A unique system number assigned by Simulink
- `block_name` is the name of the block.

The following code fragment illustrates a tag comment adjacent to a line of code generated by a Gain block at the root level of the source model:

```
/* Gain: '<Root>/UnDeadGain1' */
rtb_UnDeadGain1_h = dead_gain_U.In1 *
    dead_gain_P.UnDeadGain1_Gain;
```

The following code fragment illustrates a tag comment adjacent to a line of code generated by a Gain block within a subsystem one level below the root level of the source model:

```
/* Gain Block: <S1>/Gain */
dead_gain_B.temp0 *= (dead_gain_P.s1_Gain_Gain);
```

In addition to the tags, Real-Time Workshop documents the tags for each model in comments in the generated header file `model.h`. The following illustrates such a comment, generated from a source model, `foo`, that has a subsystem `Outer` with a nested subsystem `Inner`:

```
/* Here is the system hierarchy for this model.
 *
 * <Root> : foo
 * <S1>   : foo/Outer
 * <S2>   : foo/Outer/Inner
 */
```

There are two ways to trace code back to subsystems, blocks, and parameters in your model:

- Through HTML code generation reports via the Help browser
- By typing the appropriate `hilite_system` commands to MATLAB

When you are licensed for Real-Time Workshop Embedded Coder, the HTML report for your `model.c` file displays hyperlinks in “Regarding,” “Outport,” and other comment lines such as are shown above. Clicking such links in comments causes the associated block or subsystem to be highlighted in the model diagram. For further information, see “HTML Code Generation Reports” on page 3-37 of the Real-Time Workshop Getting Started documentation.

Using HTML reports is generally the fastest way to trace code back to the model, but when you know what you are looking for you might achieve the same result at the command line. To manually trace a tag back to the generating block using the `hilite_system` command,

- 1 Open the source model.
- 2 Close any other model windows that are open.
- 3 Use the MATLAB `hilite_system` command to view the desired system and block.

As an example, consider the model `foo` mentioned above. If `foo` is open,

```
hilite_system('<S1>')
```

opens the subsystem `Outer` and

```
hilite_system('<S2>/Gain1')
```

opens the subsystem `Outer` and selects and highlights the Gain block `Gain1` within that subsystem.

Other Interactions Between Simulink and Real-Time Workshop

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

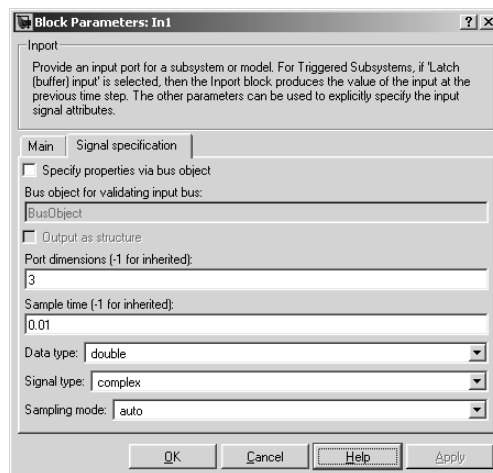
- Data type
- Line widths
- Sample times

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

You can verify what data types any given Simulink block supports by typing `showblockdatatypetable`

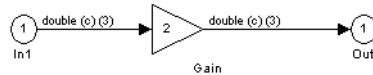
at the MATLAB prompt, or (from the Help browser) clicking the command above.

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

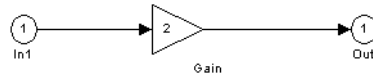
Inherited sample times in source blocks (e.g., a root inport) can sometimes lead to unexpected and unintended sample time assignments. Since a block may specify an inherited sample time, information available at the outset is often insufficient to compile a block diagram completely. In such cases, the 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, Simulink 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 according to the following rules:

- 1 If the current system has at least one rate in it, the block is assigned the fastest rate.
- 2 If no rate exists and the model is configured for a variable-step solver, the block is assigned a continuous sample time (but fixed in minor time steps). Note that Real-Time Workshop (with the exception of the rapid simulation and S-function targets) does not currently support variable-step solvers.
- 3 If no rate exists and the model is configured for a fixed-step solver, the block is assigned a discrete sample time of $(T_f - T_i)/50$, where T_i is the simulation start time and T_f is the simulation stop time. If T_f is infinity, the default sample time is set to 0.2.

To ensure a completely deterministic model (one where no sample times are set using the above rules), you should explicitly specify the sample times of all your source blocks. Source blocks include root inport blocks and any 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 default sample time is derived from destination blocks or the default rule.



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

Constant Block Sample Times. You can specify a sample time for Constant blocks. This has certain implications for code generation.

When a sample time of `inf` is selected for a Constant block,

- If **Inline parameters** is on, the block will take on a constant sample time, and propagate a constant sample time downstream.
- If **Inline parameters** is off, the constant block inherits its sample time—which will be nonconstant—and propagates that sample time downstream.

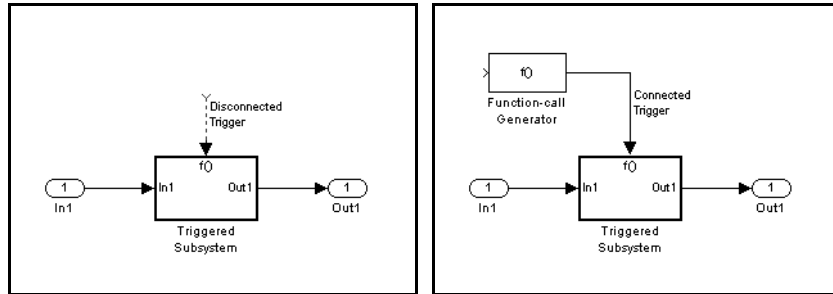
Generated code for any block differs when it has a constant sample time; its outputs are represented in the constant block outputs structure instead of in the general block outputs structure. The generated code thus reflects that the Constant block propagates a constant sample time downstream if a sample time of `inf` is specified and **Inline parameters** is on.

Block Execution Order

Once Simulink compiles the block diagram, it creates a `model.rtw` file (analogous to an object file generated from a C file). The `model.rtw` file contains all the connection information of the model, as well as the necessary signal attributes. Thus, the timing engine in Real-Time Workshop can determine when blocks with different rates should be executed.

You cannot override this execution order by directly calling a block (in hand-written code) in a model. For example, the `disconnected_trigger` model below has its trigger port connected to ground, which can lead to all blocks

inheriting a constant sample time. Calling the trigger function, $f()$, directly from user code does not work correctly and should never be done. Instead, you should use a function-call generator to properly specify the rate at which $f()$ should be executed, as shown in the `connected_trigger` model below.



Instead of the function-call generator, you could use any other 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 Real-Time Workshop is to build individual models separately and then hand-code the I/O between the models. This approach places the burden of data consistency between models on the developer of the models. Another approach is to let Simulink and Real-Time Workshop ensure data consistency between rates and generate multirate code for use in a multitasking environment. The Real-Time Workshop interrupt template and VxWorks support libraries provide blocks that support both synchronous and asynchronous I/O via a double-buffering scheme. For a description of the Real-Time Workshop libraries, see “Asynchronous Support” on page 16-1. For more information on multirate code generation, see Chapter 8, “Models with Multiple Sample Rates.”

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

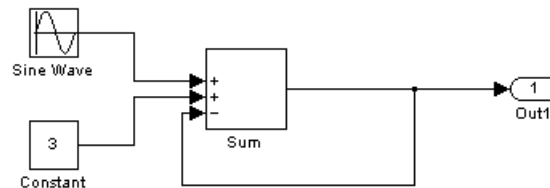
- $x = y + 2$
- $y = -x$

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, e.g., using gradient based search) or “solve” the system of equations. In the previous example, solving the system into an explicit form leads to

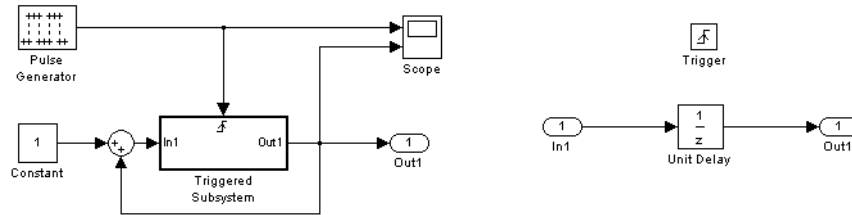
- $2x = 2$
- $y = -x$
- $x = 1$
- $y = -1$

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. Simulink is often able to solve models that contain algebraic loops, such as the diagram shown below.



Real-Time Workshop does not produce code that solves algebraic loops. This restriction includes models that use Algebraic Constraint blocks in feedback paths. However, Simulink can often eliminate all or some 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. See the Simulink documentation for details.

Algebraic Loops in Triggered Subsystems. While Simulink can minimize algebraic loops involving atomic and enable subsystems, a special consideration applies to some triggered subsystems. In Simulink versions prior to 6.0, triggered subsystems always combined their output and update methods in simulation and in generated code. This led to cases where Simulink would report algebraic loops with triggered and enabled-triggered subsystems even when the loops consisted of blocks that did not have direct feedthrough. An example for which code can be generated is shown in the model and triggered subsystem below:



The default behavior of Simulink 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 Simulink to solve the problem by splitting the output and update methods of triggered and enabled-triggered subsystems when necessary and feasible. If you want Real-Time Workshop to take advantage of this feature, select the **Minimize algebraic loop occurrences** check box in the **Subsystem parameters** dialog. Select this option to avoid algebraic loop warnings in triggered subsystems involved in loops. Note that if you always 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 note that if a direct feedthrough block (such as a Gain block) is connected to the inport in the above triggered subsystem, Simulink cannot solve the problem, and Real-Time Workshop is unable to generate code.

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

Selecting a Target Configuration

The process of generating target-specific code is controlled by three things:

- A system target file
- A template makefile
- A make command

The System Target File Browser lets you specify such a configuration in a single step, choosing from a wide variety of ready-to-run configurations.

The System Target File Browser

To select a target configuration using the System Target File Browser,

- 1** Click **Real-Time Workshop** on the **Configuration Parameters** dialog box. The **Real-Time Workshop** pane appears.
- 2** Click the **Browse** button next to the **RTW System target file** field. This opens the System Target File Browser. The browser displays a list of all currently available target configurations, including customizations. When you select a target configuration, Real-Time Workshop automatically chooses the appropriate system target file, template makefile, and make command.

Figure 2-14 shows the System Target File Browser with the generic real-time target selected.

- 3** Click the desired entry in the list of available configurations. The background of the list box turns yellow to indicate an unapplied choice has been made. To apply it, click **Apply** or **OK**.

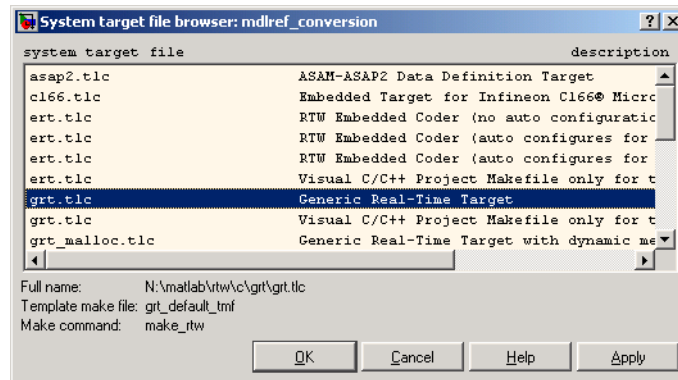


Figure 2-14: The System Target File Browser

When you choose a target configuration, Real-Time Workshop automatically chooses the appropriate system target file, template makefile, and make command for the selected target, and displays them in the RTW System File edit field. The Description of the target file from the browser is placed below its name in the general **Real-Time Workshop** pane.

Available Targets

The following table lists supported system target files and their associated code formats. The table also gives references to relevant manuals or chapters in this book. All of these targets are built using the `make_rtw` make command.

Note You can select any target of interest using the System Target File Browser. This allows you to experiment with configuration options and save your model with different configurations. However, you will not be able to build or generate code for non-GRT targets unless you have the appropriate license on your system (Real-Time Workshop Embedded Coder for ERT, Real-Time Windows for RTWIN, etc.).

Each system target file invokes one or more template makefiles. The template makefile that is invoked activates a particular compiler (e.g., `lcc`, `gcc`, Watcom, or Borland). This is specified for you by `MEXOPTS`, which is determined when you

run `mex -setup` to select a compiler for `mex`. The one exception is the Visual C/C++ compiler, which has system target file browser entries.

Targets Available from the System Target File Browser

Target/Code Format	System Target File	Template Makefile /Comments	Relevant Chapters
Real-Time Workshop Embedded Coder (PC or UNIX)	<code>ert.tlc</code>	<code>ert_default_tmf</code> Use <code>mex -setup</code> to configure for <code>lcc</code> , <code>watcom</code> , <code>borland</code> , <code>vc</code> , <code>gcc</code> , etc.	Real-Time Workshop Embedded Coder documentation
Real-Time Workshop Embedded Coder for Visual C/C++ Project Makefile	<code>ert.tlc</code>	<code>ert_msvc.tmf</code>	Real-Time Workshop Embedded Coder documentation
Real-Time Workshop Embedded Coder for Tornado (VxWorks)	<code>ert.tlc</code>	<code>ert_tornado.tmf</code>	Real-Time Workshop Embedded Coder documentation
Generic Real-Time for PC/UNIX	<code>grt.tlc</code>	<code>grt_default_tmf</code> Use <code>mex -setup</code> to configure for <code>lcc</code> , <code>watcom</code> , <code>borland</code> , <code>vc</code> , <code>gcc</code> , etc.	Chapter 3
Generic Real-Time for Visual C/C++ Project Makefile	<code>grt.tlc</code>	<code>grt_msvc.tmf</code>	Chapter 3
Generic Real-Time (dynamic) for PC/UNIX	<code>grt_malloc.tlc</code>	<code>grt_malloc_default_tmf</code> Use <code>mex -setup</code> to configure for <code>lcc</code> , <code>watcom</code> , <code>borland</code> , <code>vc</code> , <code>gcc</code> , etc.	Chapter 3

Targets Available from the System Target File Browser (Continued)

Target/Code Format	System Target File	Template Makefile /Comments	Relevant Chapters
Generic Real-Time (dynamic) for Visual C/C++ Project Makefile	grt_malloc.tlc	grt_malloc_msvc.tmf	Chapter 3
Rapid Simulation Target (default for PC or UNIX)	rsim.tlc	rsim_default_tmf	Chapter 12
Rapid Simulation Target for Watcom	rsim.tlc	rsim_watc.tmf	Chapter 12
Rapid Simulation Target for Visual C/C++	rsim.tlc	rsim_vc.tmf	Chapter 12
Rapid Simulation Target for Borland	rsim.tlc	rsim_bc.tmf	Chapter 12
Rapid Simulation Target for LCC	rsim.tlc	rsim_lcc.tmf	Chapter 12
Rapid Simulation Target for UNIX	rsim.tlc	rsim_unix.tmf	Chapter 12
S-Function Target for PC or UNIX	rtwsfcn.tlc	rtwsfcn_default_tmf	Chapter 11
S-Function Target for Watcom	rtwsfcn.tlc	rtwsfcn_watc.tmf	Chapter 11
S-Function Target for Visual C/C++	rtwsfcn.tlc	rtwsfcn_vc.tmf	Chapter 11
S-Function Target for Borland	rtwsfcn.tlc	rtwsfcn_bc.tmf	Chapter 11

Targets Available from the System Target File Browser (Continued)

Target/Code Format	System Target File	Template Makefile /Comments	Relevant Chapters
S-Function Target for LCC	rtwsfcn.tlc	rtwsfcn_lcc.tmf rtwsfcn_unix.tmf	Chapter 11
Tornado (VxWorks) Real-Time Target	tornado.tlc	tornado.tmf	Chapter 13
ASAM-ASAP2 Data Definition Target	asap2.tlc	asap2_default_tmf	Appendix B
Windows Real-Time Target for Watcom	rtwin.tlc	win_watc.tmf	Real-Time Windows Target documentation
Windows Real-Time Target for Visual C/C++	rtwin.tlc	win_vc.tmf	Real-Time Windows Target documentation
xPC Target for Watcom C/C++ or Visual C/C++	xpctarget.tlc	xpc_default_tmf xpc_vc.tmf xpc_watc.tmf	xPC Target documentation
Embedded Target for the TI TMS320C2000 DSP Platform	ti_c2000_grt.tlc ti_c2000_ert.tlc	ti_c2000_grt.tmf ti_c2000_ert.tmf	Embedded Target for the TI TMS320C2000 DSP Platform documentation
Embedded Target for the TI TMS320C6000 DSP Platform	ti_c6000.tlc (GRT) ti_c6000_ert.tlc	ti_c6000.tmf ti_c6000_ert.tmf	Embedded Target for the TI TMS320C6000 DSP Platform documentation
Embedded Target for OSEK/VDX	osekworks.tlc proosek.tlc	osek_default_tmf	Embedded Target for OSEK/VDX documentation

Targets Available from the System Target File Browser (Continued)

Target/Code Format	System Target File	Template Makefile /Comments	Relevant Chapters
Embedded Target for Motorola MPC555	mpc555exp.tlc	mpc555exp.tmf	Embedded Target for Motorola MPC555 documentation
	mpc555pil.tlc	mpc555exp_diab.tmf mpc555pil.tmf	
	mpc555rt.tlc	mpc555pil_diab.tmf mpc555rt.tmf	
Embedded Target for Motorola HC12	hc12.tlc	hc12_default_tmf	Embedded Target for Motorola HC12 documentation
Embedded Target for Infineon C166 Microcontrollers	c166.tlc	c166.tmf	Embedded Target for Infineon C166 Microcontrollers documentation

Creating Custom Targets

You can create your own system target files to build custom targets that interface with external code or operating environments. If you have in the past created system target files, note that the form of callbacks has changed between versions 5 and 6 of Real-Time Workshop. See “System Target Files” in the Real-Time Workshop Embedded Coder targeting documentation for full details, including how to make your custom targets appear in the System Target File browser and display appropriate controls in **Configuration Parameters** dialog panes.

Making an Executable

Real-Time Workshop generates code into a set of source files that vary little among different targets. Not all possible files will be generated for every model. Some files are only created when the model includes subsystems, calls external interfaces, or uses particular types of data.

Real-Time Workshop handles most of the code formatting decisions (such as identifier construction and code packaging) in consistent ways. This section describes how symbols are constructed and how generated code is allocated to source files.

Construction of Symbols

For GRT, GRT-malloc and RSim targets, Real-Time Workshop automatically constructs identifiers for variables and functions in the generated code. These symbols identify

- Signals and parameters that have Auto storage class
- Subsystem function names that are not user defined
- All Stateflow names

The components of a generated symbol include

- The root model name, followed by
- The name of the generating object (signal, parameter, state, etc.), followed by
- A unique *name mangling* string

The name mangling string is conditionally generated only when necessary to resolve potential conflicts with other generated symbols.

The length of generated symbols is limited by the **Maximum identifier length** parameter specified on the **Symbols** pane of the **Configuration Parameters** dialog. When there is a potential name collision between two symbols, a name mangling string is generated. The string 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 in general. One way to do this is to avoid using default block names (for example, Gain1, Gain2. . .) when there are many blocks of the same type in the model. Also, whenever possible, make subsystems atomic and reusable.
- Where possible, increase the **Maximum identifier length** parameter to accommodate the length of the symbols you expect to generate.

Maximum identifier length must be the same for both top and referenced models. Model referencing can involve additional naming constraints. For information, see “Symbols Options” on page 2-38 and “Parameterizing Referenced Models” on page 4-21.

Users of Real-Time Workshop Embedded Coder have additional flexibility over how symbols are constructed, via a **Symbol format** field that controls the symbol formatting in much greater detail. See the “Code Generation Options and Optimizations” chapter in the Real-Time Workshop Embedded Coder documentation for further information.

Generated Source Files and File Dependencies

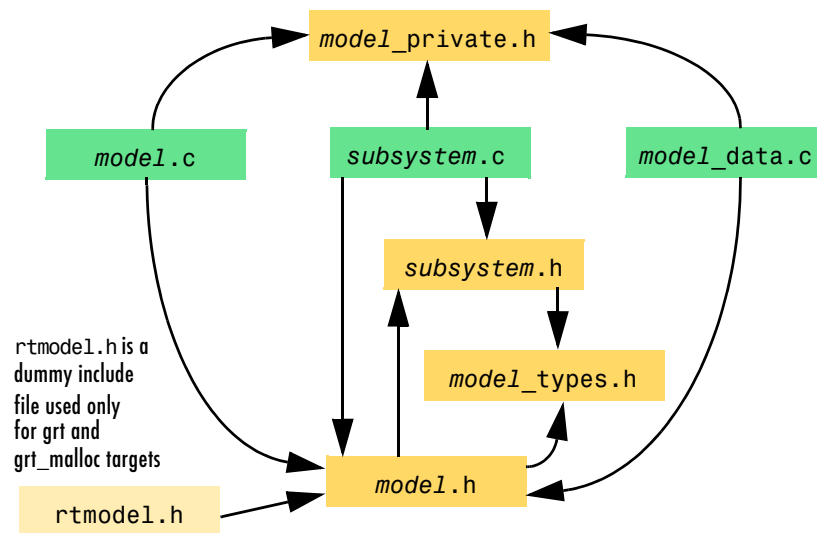
The source and make files created by Real-Time Workshop are generated into your build directory, which is created or reused in your current directory. Some files are unconditionally generated, while the existence of others depend on target settings and options (for example, support files for C-API or external mode). The generated files are listed and described in “Files and Directories Created by the Build Process” on page 2-22.

Note The file packaging of Real-Time Workshop Embedded Coder targets differs slightly (but significantly) from the file packaging described below. See the “Data Structures and Code Modules” section in the Real-Time Workshop Embedded Coder documentation for further information.

Real-Time Workshop generated source file dependencies are depicted in the following figure. Arrows coming from a file point to files it includes. As the illustration notes, other dependencies exist, for example on Simulink header files files `tmw_types.h`, `simstruc_types.h`, and optionally on `rtlibsrc.h`, plus C library files. The diagram only maps inclusion relations between files that are generated in the build directory. Utility and model reference code located

in a project directory might also be referenced by these files. See “Project Directory Structure for Model Reference Targets” on page 4-27 for details.

The diagram shows that parent system header files (*model.h*) include all child subsystem header files (*subsystem.h*). In more layered models, subsystems similarly include their children’s header files, on down the model hierarchy. As a consequence, subsystems are able to recursively “see” into all their descendants’ subsystems, as well as to see into the root system (because every *subsystem.c* includes *model.h* and *model_private.h*).



NOTE Files *model.h*, *model_private.h*, and *subsystem.h* also depend on Simulink header file *rtwtypes.h*, and conditionally on *rtlibsrc.h*. Non-ERT-based targets can have additional dependencies on *tmw_types.h* and *simstruct_types.h*.

Real-Time Workshop Generated File Dependencies

Dependencies of the Model’s Generated code

Real-Time Workshop can directly build standalone executables for the host system such as when using the GRT Target. There are also several processor and OS-specific targets which provide automated builds using a cross-compiler. All of these targets are typically makefile-based interfaces for which Real-Time Workshop provides a “Template MakeFile (TMF) to makefile”

conversion capability. Part of this conversion process is to include in the generated makefile all of the source file, header file and library file information needed (the dependencies) for a successful compilation.

In other instances, the generated model code needs to be integrated into a specific application. Or, it may be desired to enter the generated files and any file dependencies into a configuration management system. This section discusses the various aspects of the generated code dependencies and how to determine them.

Typically, the generated code for a model will consist of a small set of files:

- *model.c*
- *model.h*
- *model_data.c*
- *model_private.h*
- *rtwtypes.h*

These are generated in the build directory for a standalone model or a subdirectory under the `slprj` directory for a model reference target. There is also a top-level `main.c` file which calls the top-level model functions to execute the model. `main.c` is a static (not generated) file (such as `grt_main.c` for GRT-based targets), and is either a static file (`ert_main.c`) or is dynamically generated for ERT based targets.

The files described above will also have dependencies on other files, these dependencies occur due to:

- Including other headers files
- Using macros declared in other header files
- Calling functions declared in other source files
- Accessing variables declared in other source files

These dependencies are introduced for a number of reasons such as:

- Blocks in a model will generate code that make function calls. This can occur in several forms:
 - The called functions are declared in other source files. In some cases such as a blockset, these source file dependencies are typically managed by compiling them into a library file.

- In other cases, the called functions are provided by the compilers own runtime library, such as for functions in the ANSI-C header, `math.h`.
- Some function dependencies are themselves generated files. Some examples are for Fixed-Point utilities and nonfinite support. These dependencies are referred to as Shared Utilities. The generated functions can appear in files in the build directory for standalone models or in the `_sharedutils` directory under the `slprj` directory for builds that involve model reference.
- Models with continuous time will require solver source code.
- Real-Time Workshop options such as external Mmode, C-API, MAT-File logging are examples which will trigger additional dependencies.
- Specifying custom code can introduces dependencies.

Collecting the Dependencies. Real-Time Workshop provides several mechanisms for providing file dependency information. These mechanisms are inputs into the Real-Time Workshop build process.

- User S-Functions and blocksets
 - Directories that contain S-Function MEX-files used by a model are added to the header include path.
 - Makefile rules are created for these directories to allow source code to be found.
 - For S-functions which are not inlined with a TLC file, the S-function source file name is added to the list of sources to compile.
 - The S-Function block parameter `SFunctionModules` provides the ability to specify additional source file names, object files or precompiled library names.
 - The `rtwmakecfg.m` mechanism provides further capability in specifying dependencies. See the Developing Embedded Target Guide in the Real-Time Workshop Embedded Coder documentation for more information on the `rtwmakecfg.m` mechanism.
- S-Function Builder block provides its own GUI for specifying dependency information.
- The **Custom Code** pane of the **Real-Time Workshop Configuration Parameters** dialog provides ability to specify additional libraries, source files and include directories.

- The make variables `USER_SRCS` and `USER_INCLUDES` can be specified to provide additional user sources and additional include paths.
- The TLC phase provides the ability to specify additional dependencies using the TLC functions: `LibAddToCommonIncludes()` and `LibAddToModelSources()`. Real-Time Workshop Embedded Coder also provides TLC-based customization template capability for generating additional source files.

Makefile Considerations. As previously mentioned, Real-Time Workshop targets are typically makefile based and Real-Time Workshop provides a “Template MakeFile (TMF) to makefile” conversion capability. The Template MakeFile (TMF) contains a token expansion mechanism in which the build process will expand different tokens in the makefile to include additional dependency information. The resulting makefile contains the complete dependency information. See the Developing Embedded Target Guide in the Real-Time Workshop Embedded Coder documentation for more information on working with Template MakeFiles.

The generated makefile will contain the following information:

- Names of the source file dependencies (via various SRC variables)
- Directories where source files are located (via unique rules)
- Location of the header files (via the INCLUDE variables)
- Precompiled library dependencies (via the LIB variables)
- Libraries which need to be compiled and created (via rules and the LIB variables)

A property of make utilities is that the specific location for a given source C file does not need to be specified. If there is a rule for that directory and the source file name is a prerequisite in the makefile, the make utility can find the source file and compile it. Similarly, the C compiler (preprocessor) does not require absolute paths to the headers. Given the name of header file via a `#include` directive and an include path, it is able to find the header. The generated C source code depends on this standard compiler capability.

Also, libraries are typically created and linked against, but occlude the specific functions that are being used.

Although the build process is successful and can create a minimum-size executable, these properties can make it difficult to manually determine the

minimum list of file dependencies along with their fully qualified paths. The makefile used can be a starting point to determining the dependencies that the generated model code has.

An additional approach to determining the dependencies is by using linker information, such as a linker map file, to determine the symbol dependencies. The location of Real-Time Workshop and blockset source and header files is provided below to assist in locating the dependencies.

Real-Time Workshop Static File Dependencies. There are a several locations within the MATLAB directory tree that contain the Real-Time Workshop-specific static file dependencies:

- *matlabroot/rtw/c/libsrc/*

This directory contains many functions on which the generated code can be dependent. The directory contains a *rtwmakecfg.m* file which is used to create the appropriate rules and variables in the generated makefile. The files in this directory are compiled into a library. Note that for host based targets, Real-Time Workshop provides a precompiled library for the functions in this directory. If the optimization options (*OPT_OPTS*) are changed for a Real-Time Workshop build, the makefile will recompile the library; otherwise the precompiled library is used.

- *matlabroot/rtw/c/src/*

This directory has subdirectories and contains additional files that may need to be compiled. Examples include solver functions (for continuous time support), external mode support files, C-API support files and S-function support files. Source files in this directory are included into the build process using in the *SRC* variables of the makefile.

- *matlabroot/rtw/extern/include/*.h*
- *matlabroot/simulink/include/*.h*

These directories contain additional header file dependencies such as *tmwtypes.h*, *simstruc_types.h*, and *simstruc.h*.

Note For ERT-based targets, several header dependencies from the above locations can be avoided. ERT based targets will generate the minimum necessary set of typedefs, macros, etc., in the file *rtwtypes.h*.

Blockset Static File Dependencies. Blockset products leverage the `rtwmakecfg.m` mechanism to provide Real-Time Workshop with dependency information. As such, the `rtwmakecfg.m` file provided by the blockset contains the listing of include path and source path dependencies for the blockset. Typically, blocksets create a library from the source files which the generated model code can then link against. The libraries are created and identify using the `rtwmakecfg.m` mechanism, similar to RTW's `libsrc` directory. The locations of the `rtwmakecfg.m` files for the blocksets are:

- `matlabroot/commblocks/commblocksdemos/rtwmakecfg.m`
- `matlabroot/commblocks/commex/rtwmakecfg.m`
- `matlabroot/dspblocks/dspmex/rtwmakecfg.m`
- `matlabroot/fuzzy/fuzzy/rtwmakecfg.m`
- `matlabroot/physmod/drive/drive/rtwmakecfg.m`
- `matlabroot/physmod/mech/mech/rtwmakecfg.m`
- `matlabroot/physmod/powersys/powersys/rtwmakecfg.m`

If the model being compiled uses one or more of these blocksets, you can determine directory and file dependency information from the respective `rtwmakecfg.m` file.

Compilation and Linking

After completing code generation, the build process determines whether or not to continue and compile and link an executable program. This decision is governed by the following parameters:

- **Generate code only** option
When this option is selected, the build process always omits the make phase.
- **Makefile-only target**
The Visual C/C++ Project Makefile versions of the `grt`, `grt_malloc`, and Real-Time Workshop Embedded Coder target configurations generate a Visual C/C++ project makefile (`model.mak`). To build an executable, you must open `model.mak` in the Visual C/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 HOST variable can take on one of three possible values: PC, UNIX, or ANY.

By default, HOST is set to UNIX in template makefiles designed for use with UNIX (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 Simulink 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 Simulink 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
```

Choosing and Configuring Your Compiler

The Real-Time Workshop build process depends upon the correct installation of one or more supported compilers. Note that *compiler*, in this context, refers to a development environment containing a linker and make utility, in addition to a high-level language compiler.

The build process also requires the selection of a template makefile. The template makefile determines which compiler is run, during the make phase of the build, to compile the generated code.

This section discusses how to install a compiler and choose an appropriate template makefile, on both Windows and UNIX systems.

To determine which template makefiles are appropriate for your compiler and target, see Table, Targets Available from the System Target File Browser.

Choosing and Configuring Your Compiler on Windows

On Windows, you must install one or more supported compilers. In addition, you must define an environment variable associated with each compiler. Make sure that your compiler is installed as described in “Third-Party Compiler Installation on Windows” on page 1-12.

You can select a template makefile that is specific to your compiler. For example, `grt_bc.tmf` designates the Borland C/C++ compiler, and `grt_vc.tmf` designates the Visual C/C++ compiler.

Alternatively, you can choose a default template makefile that will select the default compiler for your system. The default compiler is the compiler MATLAB uses to build MEX-files. You can set up the default compiler by using the MATLAB `mex` command as shown below.

```
mex -setup
```

See “External Interfaces/API” in the MATLAB documentation for information on the `mex` command.

Default template makefiles are named `target_default_tmf`. For example, the default template makefile for the generic real-time (GRT) target is `grt_default_tmf`.

The build process uses the following logic to locate a compiler for the generated code:

- 1 If a specific compiler is named in the template makefile, the build process uses that compiler.
- 2 If the template makefile designates a default compiler (as in `grt_default_tmf`), the build process uses the same compiler as those used for building C MEX-files.
- 3 If no default compiler is established, the build process examines environment variables that define the path to installed compilers, and selects the first compiler located. The variables are searched in the following order:
 - `MSDevDir` or `DEVSTUDIO` (defining the path to the Microsoft Visual C/C++)
 - `WATCOM` (defining the path to the Watcom C/C++ compiler)
 - `BORLAND` (defining the path to the Borland C/C++ compiler)
- 4 If none of the above environment variables is defined, the build process selects the `lcc` compiler, which is shipped and installed with MATLAB.

Compile/Build Options for Visual C/C++ and Intel Compilers

Real-Time Workshop offers two sets of template makefiles designed for use with Visual C/C++ and the Intel compiler sometimes used with Visual C/C++.

To run these compilers and build an executable within the Real-Time Workshop build process, use one of the `target_vc.tmf` (or `target_intel.tmf`) template makefiles:

- `ert_vc.tmf` (`ert_intel.tmf`)
- `grt_malloc_vc.tmf` (`grt_malloc_intel.tmf`)
- `grt_vc.tmf` (`grt_intel.tmf`)
- `rsim_vc.tmf` (`rsim_intel.tmf`)

Alternatively, you can choose to create a project makefile (`model.mak`) suitable for use with the Visual C/C++ IDE. In this case, you must compile and link your code within the Visual C/C++ environment. To create a Visual C/C++ project makefile, choose one of the Visual C/C++ Project Makefile versions of the `grt`, `ert`, or `grt_malloc` target configurations. These configurations use the `target_msvc.tmf` template makefiles:

- `ert_msvc.tmf`
- `grt_malloc_msvc.tmf`
- `grt_msvc.tmf`

Choosing and Configuring Your Compiler on UNIX

On UNIX, the Real-Time Workshop build process uses the default compiler. `cc` is the default on all platforms except SunOS, where `gcc` is the default.

You should choose the UNIX-specific template makefile that is appropriate to your target. For example, `grt_unix.tmf` is the correct template makefile to build a generic real-time program under UNIX.

Including S-function Source Code

When Real-Time Workshop builds models with S-functions, source code for the S-functions can either be in the current directory or in the same directory as their MEX-file. Real-Time Workshop adds an include path to the generated makefiles whenever it finds a file named `<s-function-name>.h` in the same directory that the S-function MEX-file is in. This directory must be on the MATLAB path.

Similarly, Real-Time Workshop will add a rule for the directory when it finds a file `<s-function-name>.c` (or `.cpp`) in the same directory as the S-function MEX-file is in.

There is therefore no need to copy the S-function source file into the MATLAB current directory or to create an `rtwmakecfg.m` file in the S-function's directory, as was sometimes necessary prior to version 6.

Template Makefiles and Make Options

Real-Time Workshop includes a set of built-in template makefiles that are designed to build programs for specific targets.

There are two types of template makefiles:

- *Compiler-specific* template makefiles are designed for use with a particular compiler or development system.

By convention, compiler-specific template makefiles are named according to the target and compiler (or development system). For example, `grt_vc.tmf` is the template makefile for building a generic real-time program under Visual C/C++; `ert_lcc.tmf` is the template makefile for building a Real-Time Workshop Embedded Coder program under the LCC compiler.

- *Default* template makefiles make your model designs more portable, by choosing the correct compiler-specific makefile and compiler for your installation. “Choosing and Configuring Your Compiler” on page 2-68 describes the operation of default template makefiles in detail.

Default template makefiles are named *target_default_tmf*. 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 a Real-Time Workshop Embedded Coder program.

You can supply options to makefiles via arguments to the **Make command** field in the general **Real-Time Workshop** pane of the **Configuration Parameters** dialog. Append the arguments after `make_rtw` (or `make_xpc` 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.

Compiler-Specific Template Makefiles

This section documents the available compiler-specific template makefiles and common options you can use with each. Complete details on the structure of template makefiles are provided in the Real-Time Workshop Embedded Coder documentation. This information is provided for those who want to customize template makefiles.

Template Makefiles for UNIX

- `ert_unix.tmf`
- `grt_malloc_unix.tmf`
- `grt_unix.tmf`
- `rsim_unix.tmf`
- `rtwsfcn_unix.tmf`

The template makefiles for UNIX platforms are designed to be used with GNU Make. These makefile are set up to conform to the guidelines specified in the IEEE Std 1003.2-1992 (POSIX) standard.

You can supply options via arguments to the make command.

- `OPTS` — User-specific options, for example,
`make_rtw OPTS="-DMYDEFINE=1"`
- `OPT_OPTS` — Optimization options. Default is `-O`. To enable debugging specify 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 needed by S-functions.
- `USER_INCLUDES` — Additional include paths, for example,
`USER_INCLUDES="-Iwhere-ever -Iwhere-ever2")`

These options are also documented in the comments at the head of the respective template makefiles.

Template Makefiles for Visual C/C++

Real-Time Workshop offers two sets of template makefiles designed for use with Visual C/C++.

To build an executable within the Real-Time Workshop build process, use one of the `target_vc.tmf` template makefiles:

- `ert_vc.tmf`
- `grt_malloc_vc.tmf`
- `grt_vc.tmf`
- `rsim_vc.tmf`
- `rtwsfcn_vc.tmf`

You can supply options via arguments to the make command.

- `OPT_OPTS` — Optimization option. Default is `-O2`. To enable debugging specify as `OPT_OPTS=-Zd`.
- `OPTS` — User-specific options.
- `CPP_OPTS` — C++ compiler options.
- `USER_SRCS` — Additional user sources, such as files needed by S-functions.
- `USER_INCLUDES` — Additional include paths, for example,

```
USER_INCLUDES=" -Iwhere-ever -Iwhere-ever2"
```

These options are also documented in the comments at the head of the respective template makefiles.

Visual C/C++ Code Generation Only. To create a Visual C/C++ project makefile (*model.mak*) without building an executable, use one of the *target_msvc.tmf* template makefiles:

- `ert_msvc.tmf`
- `grt_malloc_msvc.tmf`
- `grt_msvc.tmf`

These template makefiles are designed to be used with `nmake`, which is bundled with Visual C/C++.

You can supply the following options via arguments to the `nmake` command:

- `OPTS` — User-specific options, for example,

```
make_rtw OPTS="/D MYDEFINE=1"
```
- `USER_SRCS` — Additional user sources, such as files needed by S-functions.
- `USER_INCLUDES` — Additional include paths, for example,

```
USER_INCLUDES=" -Iwhere-ever -Iwhere-ever2")
```

These options are also documented in the comments at the head of the respective template makefiles.

Template Makefiles for Watcom C/C++

Note As of this printing, the Watcom C compiler is no longer available from the manufacturer. Real-Time Workshop continues to ship with Watcom-related template makefiles at this time. However, this policy may be subject to change in the future.

- `drt_watc.tmf`
- `ert_watc.tmf`
- `grt_malloc_watc.tmf`
- `grt_watc.tmf`
- `rsim_watc.tmf`
- `rtwsfcn_watc.tmf`
- `win_watc.tmf`

Real-Time Workshop provides template makefiles to create an executable for Windows using Watcom C/C++. These template makefiles are designed to be used with `wmake`, which is bundled with Watcom C/C++.

You can supply options via arguments to the `make` command. Note that the location of the quotes is different from the other compilers and `make` utilities discussed in this chapter.

- `OPTS` — User-specific options, for example,
`make_rtw "OPTS=-DMYDEFINE=1"`
- `OPT_OPTS` — Optimization options. The default optimization option is `-oxat`. To turn off optimization and add debugging symbols, specify the `-d2` compiler switch in the `make` command, for example,
`make_rtw "OPT_OPTS=-d2"`
- `CPP_OPTS` — C++ compiler options.
- `USER_OBJS` — Additional user objects, such as files needed by S-functions.
- `USER_PATH` — The directory path to the source (`.c`) files that are used to create any `.obj` files specified in `USER_OBJS`. Multiple paths must be separated with a semicolon. For example,
`USER_PATH=path1;path2"`

- `USER_INCLUDES` — Additional include paths, for example,

```
USER_INCLUDES=-Iinclude-path1 -Iinclude-path2
```

These options are also documented in the comments at the head of the respective template makefiles.

Template Makefiles for Borland C/C++

- `ert_bc.tmf`
- `grt_bc.tmf`
- `grt_malloc_bc.tmf`
- `rsim_bc.tmf`
- `rtwsfcn_bc.tmf`

Real-Time Workshop provides template makefiles to create an executable for Windows using Borland C/C++.

You can supply these options via arguments to the make command:

- `OPTS` — User-specific options, for example,

```
make_rtw OPTS="-DMYDEFINE=1"
```

- `OPT_OPTS` — Optimization options. Default is none. To turn off optimization and add debugging symbols, specify the `-v` compiler switch in the make command.

```
make_rtw OPT_OPTS="-v"
```

- `CPP_OPTS` — C++ compiler options.
- `USER_OBJS` — Additional user objects, such as files needed by S-functions.
- `USER_PATH` — The directory path to the source (`.c`) files that are used to create any `.obj` files specified in `USER_OBJS`. Multiple paths must be separated with a semicolon. For example,

```
USER_PATH="path1;path2"
```

- `USER_INCLUDES` - Additional include paths, for example,

```
USER_INCLUDES="-Iinclude-path1 -Iinclude-path2")
```

These options are also documented in the comments at the head of the respective template makefiles.

Template Makefiles for LCC

- `ert_lcc.tmf`
- `grt_lcc.tmf`
- `grt_malloc_lcc.tmf`
- `rsim_lcc.tmf`
- `rtwsfcn_lcc.tmf`

Real-Time Workshop provides template makefiles to create an executable for Windows using LCC compiler Version 2.4 and GNU Make (`gmake`).

You can supply options via arguments to the `make` command:

- `OPTS` — User-specific options, for example,
`make_rtw OPTS="-DMYDEFINE=1"`
- `OPT_OPTS` — Optimization options. Default is none. To enable debugging, specify `-g4` in the `make` command:
`make_rtw OPT_OPTS="-g4"`
- `CPP_OPTS` — C++ compiler options.
- `USER_SRCS` — Additional user sources, such as files needed by S-functions.
- `USER_INCLUDES` — Additional include paths, for example,
`USER_INCLUDES="-Iwhere-ever -Iwhere-ever2")`

For LCC, have a `/` as file separator before the file name instead of a `\`, for example, `d:\work\proj1/myfile.c`.

These options are also documented in the comments at the head of the respective template makefiles.

Configuring the Generated Code with TLC

This section covers features of the Real-Time Workshop Target Language Compiler that help you to fine-tune your generated code. To learn more about TLC, see the Target Language Compiler Reference Guide documentation.

Target Language Compiler Variables and Options

The Target Language Compiler supports extended code generation variables and options in addition to those included in the code generation options categories of the **Real-Time Workshop** pane.

There are two ways to set TLC variables and options:

- Assigning TLC variables in the system target file
- Entering TLC options or variables in the **RTW system target file** field on the general **Real-Time Workshop** pane of the **Configuration Parameters** dialog

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*.

The %assign statement is described in the Target Language Compiler Reference Guide. We recommend that you 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 are declared by all functions in the entire model cannot exceed MaxStackSize (in bytes). N can be any positive integer. The default value for MaxStackSize is rtInf, i.e., unlimited stack size.
MaxStackVariableSize=N	When the Enable local block outputs check box is selected, this limits the size of any local variable declared in a function to N bytes, where N>0. A variable whose size exceeds MaxStackVariableSize is allocated in global, rather than local, memory.
WarnNonSaturatedBlocks = <i>value</i>	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: <ul style="list-style-type: none"> • 0 — No warning is displayed. • 1 — Displays one warning for the model during code generation • 2 — Displays one warning that contains a list of all offending blocks

Setting Target Language Compiler Options

You can enter TLC options directly into the **RTW System target file** field in the **Real-Time Workshop** tab of the **Configuration Parameters** dialog, by appending the options and arguments to the system target filename. This is equivalent to invoking the Target Language Compiler with options on the

MATLAB command line. The most common options are shown in the table below.

Target Language Compiler Options

Option	Description
<code>-Ipath</code>	Adds <i>path</i> to the list of paths in which to search for target files (.tlc files).
<code>-m[N a]</code>	Maximum number of errors to report when an error is encountered (default is 5). For example, <code>-m3</code> specifies that at most three errors will be reported. To report all errors, specify <code>-ma</code> .
<code>-d[g n o]</code>	Specifies debug mode (generate, normal, or off). Default is off. When <code>-dg</code> is specified, a .log file is create for each of your TLC files. When debug mode is enabled (i.e., generate or normal), the Target Language Compiler displays the number of times each line in a target file is encountered.
<code>-aRTWCAPI</code>	<code>-aRTWCAPI=1</code> to generate API for both signals and parameters
<code>-aRTWCAPISignals</code>	<code>-aRTWCAPISignals=1</code> to generate API for signals only
<code>-aRTWCAPIParams</code>	<code>-aRTWCAPIParams=1</code> to generate API for parameters only
<code>-aVariable=val</code>	Equivalent to the TLC statement <code>%assign Variable = val</code> Note: It is best to use %assign statements in the TLC files, rather than the <code>-a</code> option.

You can speed your TLC development cycle by not rebuilding code when your TLC files have changed, but your model has not. See “Retain .rtw File” on page 2-41 for information on how to do this.

Generated Code Formats

Introduction (p. 3-2)	Explains the concept of code formats and relationship to targets
Choosing a Code Format for Your Application (p. 3-9)	Discusses the applicability and limitations of code formats and targets with regard to types of applications
Real-Time Code Format (p. 3-13)	Describes code generation for building nonembedded applications
Real-Time malloc Code Format (p. 3-15)	Describes code generation for building nonembedded applications with dynamic allocation
S-Function Code Format (p. 3-17)	Describes code generation for building S-function targets
Embedded C Code Format (p. 3-18)	Describes code generation for building embedded applications

Introduction

Real-Time Workshop provides five different *code formats*. Each code format specifies a framework for code generation suited for specific applications.

The five code formats and corresponding application areas are

- Real-time: Rapid prototyping
- Real-time malloc: Rapid prototyping
- S-function: Creating proprietary S-function .dll or MEX-file objects, code reuse, and speeding up your simulation
- Model Reference: Creates .dll or MEX-file objects from entire models that other models can use, sometimes in place of S-functions.
- Embedded C: Deeply embedded systems

This chapter discusses the relationship of code formats to the available target configurations, and factors you should consider when choosing a code format and target. This chapter also summarizes the real-time, real-time malloc, S-function, and embedded C code formats.

Targets and Code Formats

A *target* (such as the GRT target) is an environment for generating and building code intended for execution on a certain hardware or operating system platform. A target is defined at the top level by a system target file, which in turn invokes other target-specific files.

A *code format* (such as Embedded-C or RealTime) is one property of a target. The code format controls decisions made at several points in the code generation process. These include whether and how certain data structures are generated (for example, `SimStruct` or `rtModel`), whether or not static or dynamic memory allocation code is generated, and the calling interface used for generated model functions. In general, the Embedded-C code format is more efficient than the RealTime code format. Embedded-C code format provides more compact data structures, a simpler calling interface, and static memory allocation. These characteristics make the Embedded-C code format the preferred choice for production code generation.

In prior releases, only the ERT target and targets derived from the ERT target used the Embedded-C code format. Non-ERT targets used other code formats (e.g., RealTime or RealTimeMalloc).

In Release 14, the GRT target uses the Embedded-C code format for back end code generation. This includes generation of both algorithmic model code and supervisory timing and task scheduling code. The GRT target (and derived targets) generates a RealTime code format wrapper around the Embedded-C code. This wrapper provides a calling interface that is backward compatible with existing GRT-based custom targets. The wrapper calls are compatible with the main program module of the GRT target (`grt_main.c`). Note that this use of wrapper calls incurs some calling overhead; the pure Embedded-C calling interface generated by the ERT target is more highly optimized.

The calling interface generated by the ERT target is described in the “Data Structures and Program Execution” chapter of the Real-Time Workshop Embedded Coder documentation. The calling interface generated by the GRT target is described in Chapter 7, “Program Architecture.”

Code format unification simplifies the conversion of GRT-based custom targets to ERT-based targets. See XXX for a description of target conversion issues.

Backwards Compatibility of Code Formats

Because GRT targets now use Embedded-C code format, existing applications that depend on the RealTime code format’s calling interface could have compatibility issues. To address this, a set of macros is generated (in `model.h`) that maps Embedded-C data structures to the identifiers that RealTime code format used. The following, which can be found in any `model.h` file created for a GRT target, describes these identifier mappings:

```

/* Backward compatible GRT Identifiers */
#define rtB                               model_B
#define BlockIO                           BlockIO_model
#define rtXdot                             model_Xdot
#define StateDerivatives                  StateDerivatives_model
#define tXdis                              model_Xdis
#define StateDisabled                     StateDisabled_model
#define rtY                                model_Y
#define ExternalOutputs                   ExternalOutputs_model
#define rtP                                model_P
#define Parameters                         Parameters_model

```

Since the GRT target now uses the Embedded-C code format for back end code generation, many Embedded-C optimizations are available to all Real-Time Workshop users. In general, the GRT and ERT targets now have many more

common features, but the ERT target offers additional controls for common features. The availability of features is now determined by licensing, rather than being tied to code format. The following table compares features available with a Real-Time Workshop license with those available under a Real-Time Workshop Embedded Coder license:

Comparison of Features Licensed with Real-Time Workshop vs. Real-Time Workshop Embedded Coder

Feature	Real-Time Workshop License	Real-Time Workshop Embedded Coder License
rtModel data structure	Full rtModel struct generated. GRT variable declaration: <code>rtModel_model model_M;</code>	rtModel is optimized for the model. Suppression of error status field, data logging fields, and in the struct is optional. ERT variable declaration: <code>RT_MODEL_model model_M;</code>
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 etc.) 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	Always generated	Option to suppress terminate function
Combined output/update function	Separate output/update functions are generated.	Option to generate combined output/update function

**Comparison of Features Licensed with Real-Time Workshop
vs. Real-Time Workshop Embedded Coder (Continued)**

Feature	Real-Time Workshop License	Real-Time Workshop Embedded Coder License
Optimized data initialization	Not available	Options to suppress generation of unnecessary initialization code for zero-valued memory, I/O ports, etc.
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 Real-time Workshop Embedded Coder documentation.
Target-optimized data types header file	Requires full <code>tmwtypes.h</code> header file.	Generates optimized <code>rtwtypes.h</code> header file, including only the necessary definitions required by the target.
User-defined types	User-defined types default to base types in code generation	User defined data type aliases are supported in code generation.
Simplified call interface	Non-ERT targets default to GRT interface.	ERT and ERT-based targets generate simplified interface.
Rate grouping	Not supported	Supported
Autogeneration 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.

Comparison of Features Licensed with Real-Time Workshop vs. Real-Time Workshop Embedded Coder (Continued)

Feature	Real-Time Workshop License	Real-Time Workshop Embedded Coder License
MAT-file logging	No option to suppress MAT-file logging data structures	Option to suppress MAT-file logging data structures
Reusable (multiinstance) code generation with static memory allocation	Not supported	Option to generate reusable code
Software constraint options	Support for floating point, complex, and non-finite numbers is always enabled.	Options to enable or disable support for floating-point, complex, and nonfinite number
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 via autogeneration of Simulink S-Function block
ANSI-C code generation	Supported	Supported
ISO-C code generation	Supported	Supported
GNU-C code generation	Supported	Supported
Generate scalar inlined parameters as #DEFINES	Not supported	Supported
MAT-file variable name modifier	Supported	Supported
Data exchange: C-API, External Mode, ASAP2	Supported	Supported

How Symbols Are Formatted in Generated Code

Real-Time Workshop constructs identifiers automatically for GRT targets. The symbols that are so constructed include those for

- Signals and parameters that have Auto storage class
- Subsystem function names that are not user defined
- All Stateflow names

Prior to Release 14, you could exercise these options (on the **Simulation Parameters** dialog **Code appearance** pane) to format identifiers:

- **Prefix model name to global identifiers**
- **Include System Hierarchy Number in Identifiers**
- **Include data type acronym in identifier**

These options have been removed from the Real-Time Workshop GUI and replaced by a default symbol formatting specification. The components of a generated symbol are

- The root model name, followed by
- The name of the generating object (signal, parameter, state, etc.), followed by
- A unique *name mangling* string (if required)

The number of characters that any generated symbol can have is limited by the **Maximum identifier length** parameter specified on the **Symbols** pane of the **Configuration Parameters** dialog. When there is a potential name collision between two symbols, a name mangling string is generated. The string has the minimum number of characters required to avoid the collision. The other symbol components are then inserted. If the **Maximum identifier length** parameter 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 in general. One way to do this is to avoid using default block names (for example, Gain1, Gain2. . .) when there are many blocks of the same type in the model. Also, whenever possible, make subsystems atomic and reusable.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the symbols you expect to generate. The maximum length you can specify is 256 characters.

Model Referencing Considerations. Within a model that uses model referencing, there can be no collisions between the names of the constituent models. 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 string (if needed). 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.

Note that the Real-Time Workshop Embedded Coder provides a **Symbol format** field that lets you control the formatting of generated symbols in much greater detail. See the “Code Generation Options and Optimizations” chapter in the Real-Time Workshop Embedded Coder documentation for further information.

Choosing a Code Format for Your Application

Your choice of code format is the most important code generation option. The code format specifies the overall framework of the generated code and determines its style.

When you choose a target, you implicitly choose a code format. Typically, the system target file will specify the code format by assigning the TLC variable `CodeFormat`. The following example is from `ert.tlc`.

```
%assign CodeFormat = "Embedded-C"
```

If the system target file does not assign `CodeFormat`, the default is `RealTime` (as in `grt.tlc`).

If you are developing a custom target, you must consider which code format is best for your application and assign `CodeFormat` accordingly.

Choose the `RelaTime` or `RealTime` malloc code format for rapid prototyping. If your application does not have significant restrictions in code size, memory usage, or stack usage, you might want to continue using the generic real-time (GRT) target throughout development.

For production deployment, and when your application demands that you limit source code size, memory usage, or maintain a simple call structure, the Embedded-C code format is appropriate. Consider using Real-Time Workshop Embedded Coder, if you need added flexibility to configure and optimize code.

Finally, you should choose the Model Reference or the S-function formats if you are not concerned about RAM and ROM usage and want to

- Use a model as a component, for scalability
- Create a proprietary S-function .dll or MEX-file object
- Interface the generated code using the S-function C API
- Speed up your simulation

The following table summarizes how different targets support applications:

Application	Targets
Fixed- or variable-step acceleration	RSIM, S-Function, Model Reference
Fixed-step real-time deployment	GRT, GRT-Malloc, ERT, xPC, Tornado, Real-Time Windows, TI-DSP, MPC555, ...

The following table summarizes the various options available for each Real-Time Workshop code format/target, noting exceptions below.

Features Supported by Real-Time Workshop Targets and Code Formats

Feature	GRT	Real-time malloc	RTW Embedded Coder	Tornado	S-Func	RSIM	RT Win	xPC	TI DSP	MPC 555
Static mem. allocation	X		X	X	X		X	X	X	X
Dynamic mem. allocation		X		X	X	X				
Continuous time	X	X	X	X	X	X	X	X		
C MEX S-functions (noninlined)	X	X	X	X	X	X	X	X		
Any S-function (inlined)	X	X	X	X	X	X	X	X	X	X
Minimize RAM / ROM usage			X							X

Features Supported by Real-Time Workshop Targets and Code Formats (Continued)

Feature	GRT	Real-time malloc	RTW Embedded Coder	Tornado	S-Func	RSIM	RT Win	xPC	TI DSP	MPC 555
Supports external mode	X	X	X	X		X	X	X		
Intended for rapid prototyping	X	X		X			X	X	X	
Production code			X					X	X	X ³
Batch parameter tuning and Monte Carlo methods						X				
Executes in hard real time	X ¹	X ¹	X ¹	X			X	X	X	X ²
Non real-time executable included	X	X	X			X				X
Multiple instances of a model (no Stateflow blocks in model)		X			X					X
Supports variable-step solvers					X	X				

¹The default GRT, GRT malloc, 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.

²Except MPC555 (processor-in-the-loop) and MPC555 (algorithm export) targets

³Except MPC555 (algorithm export) targets

Real-Time Code Format

The real-time code format (corresponding to the generic real-time target) is useful for rapid prototyping applications. If you want to generate real-time code while iterating model parameters rapidly, you should begin the design process with the generic real-time target. The real-time code format supports

- Continuous time
- Continuous states
- C MEX S-functions (inlined and noninlined)

For more information on inlining S-functions, see Chapter 10, “Writing S-Functions for Real-Time Workshop” and the Target Language Compiler documentation.

The real-time code format declares memory statically, that is, at compile time.

Unsupported Blocks

The real-time format does not support the following built-in user-defined blocks:

- MATLAB Fcn (note that Simulink Fcn blocks *are* supported)
- S-Function — M-file S-functions, Fortran S-functions, or C MEX S-functions that call into MATLAB (Simulink Fcn calls *are* supported)

System Target Files

- `grt.tlc` — Generic real-time target
- `rsim.tlc` — Rapid simulation target
- `tornado.tlc` — Tornado (VxWorks) real-time target

Template Makefiles

- `grt`
 - `grt_bc.tmf` — Borland C
 - `grt_vc.tmf` — Visual C
 - `grt_watc.tmf` — Watcom C

- grt_lcc.tmf — LCC compiler
- grt_unix.tmf — UNIX host
- rsim
 - rsim_bc.tmf — Borland C
 - rsim_vc.tmf — Visual C
 - rsim_watc.tmf — Watcom C
 - rsim_lcc.tmf — LCC compiler
 - rsim_unix.tmf — UNIX host
- tornado.tmf
- win_watc.tmf

Real-Time malloc Code Format

The real-time malloc code format (corresponding to the generic real-time malloc target) is very similar to the real-time code format. The differences are

- Real-time malloc declares memory dynamically.
Note that for blocks provided by The MathWorks, malloc calls are limited to the model initialization code. Generated code is designed to be free from memory leaks, provided that the model termination function is called.
- Real-time malloc allows you to deploy multiple instances of the same model with each instance maintaining its own unique data.
- Real-time malloc allows you to combine multiple models together in one executable. For example, to integrate two models into one larger executable, real-time malloc maintains a unique instance of each of the two models. If you do not use the real-time malloc format, the Real-Time Workshop will not necessarily create uniquely named data structures for each model, potentially resulting in name clashes.

`grt_malloc_main.c`, the main routine for the generic real-time malloc (`grt_malloc`) target, supports one model by default. See Chapter 17, “Combining Multiple Models” for information on modifying `grt_malloc_main` to support multiple models. `grt_malloc_main.c` is located in the directory `matlabroot/rtw/c/grt_malloc`.

Unsupported Blocks

The real-time malloc format does not support the following built-in blocks, as shown:

- Functions & Tables
 - MATLAB Fcn (note that Simulink Fcn blocks *are* supported)
 - S-Function — M-file S-functions, Fortran S-functions, or C MEX S-functions that call into MATLAB (Simulink Fcn calls *are* supported)

System Target Files

- `grt_malloc.tlc`
- `tornado.tlc` — Tornado (VxWorks) real-time target

Template Makefiles

- `grt_malloc`
 - `grt_malloc_bc.tmf` — Borland C
 - `grt_malloc_vc.tmf` — Visual C
 - `grt_malloc_watc.tmf` — Watcom C
 - `grt_malloc_lcc.tmf` — LCC compiler
 - `grt_malloc_unix.tmf` — UNIX host
- `tornado.tmf`

S-Function Code Format

The S-function code format (corresponding to the S-Function Target) generates code that conforms to the Simulink C MEX S-function API. Using the S-Function Target, you can build an S-function component and use it as an S-Function block in another model.

The S-function code format is also used by the Simulink Accelerator to create the Accelerator MEX-file.

In general you should not use the S-function code format in a system target file. However, you might need to do special handling in your inlined TLC files to account for this format. You can check the TLC variable `CodeFormat` to see if the current target is a MEX-file. If `CodeFormat = "S-Function"` and the TLC variable `Accelerator` is set to 1, the target is a Simulink Accelerator MEX-file.

See Chapter 11, “The S-Function Target,” for further information.

Embedded C Code Format

The embedded C code format corresponds to the Real-Time Workshop Embedded Coder target (ERT), and targets derived from ERT. This code format includes a number of memory-saving and performance optimizations. See the Real-Time Workshop Embedded Coder documentation for full details.

Making GRT-Based Targets ERT-Compatible

If you have developed a GRT-based custom target, it is simple to make your target ERT compatible. By doing so, you can take advantage of many efficiencies.

There are several approaches to ERT compatibility:

- If your installation is not licensed for Real-Time Workshop Embedded Coder, you can convert a GRT-based target as described in “Converting Your Target to Use `rtModel`” on page 3-19. This enables your custom target to support all current GRT features, including back end Embedded-C code generation.
- You can create an ERT-based target, but continue to use your customized version of `grt_main.c` module. To do this, you can configure the ERT target to generate a GRT-compatible calling interface, as described in “Generating GRT Wrapper Code from the ERT target” on page 3-20. This lets your target support the full ERT feature set, without changing your GRT-based run time interface. This approach requires that your installation be licensed for Real-Time Workshop Embedded Coder.
- If your installation is licensed for Real-Time Workshop Embedded Coder, you can reimplement your custom target as a completely ERT-based target, including use of an ERT generated main program. This approach lets your target support the full ERT feature set, without the overhead caused by wrapper calls.

Note If you intend to use custom storage classes (CSCs) with a custom target, you must use an ERT-based target. See the “Custom Storage Classes” chapter in the Real-Time Workshop Embedded Coder documentation for detailed information on CSCs.

For details on how GRT targets are made call-compatible with previous versions of Real-Time Workshop, see “The Real-Time Model Data Structure” on page 7-30.

Converting Your Target to Use `rtModel`

The real-time model data structure (`rtModel`) encapsulates model-specific information in a much more compact form than the `SimStruct`. Many ERT-related efficiencies 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 and parameter monitoring
- Pruning the data structure to minimize its size (ERT-derived targets only)

To take advantage of such efficiencies, you must update your GRT-based target to use the `rtModel` (unless you already did so for Release 13). The conversion requires changes to your system target file, template makefile, and main program module.

The following changes to the system target file and template makefile are required to use `rtModel` instead of `SimStruct`:

- In the system target file, add the following global variable assignment:


```
%assign GenRTModel = TLC_TRUE
```
- In the template makefile, define the symbol `USE_RTMODEL`. See one of the GRT template makefiles for an example.

The following changes to your main program module (i.e., your customized version of `grt_main.c`) are required to use `rtModel` instead of `SimStruct`:

- Include `rtmodel.h` instead of `simstruc.h`.
- Since the `rtModel` data structure has a type that includes the model name, define the following macros at the top of the main program file:


```
#define EXPAND_CONCAT(name1,name2) name1 ## name2
#define CONCAT(name1,name2) EXPAND_CONCAT(name1,name2)
#define RT_MODEL CONCAT(MODEL,_rtModel)
```

- Change the extern declaration for the function that creates and initializes the SimStruct to

```
extern RT_MODEL *MODEL(void);
```
- Change the definitions of `rt_CreateIntegrationData` and `rt_UpdateContinuousStates` to be as shown in the Release 14 version of `grt_main.c`.
- Change all function prototypes to have the argument 'RT_MODEL' instead of the argument 'SimStruct'.
- The prototypes for the functions `rt_GetNextSampleHit`, `rt_UpdateDiscreteTaskSampleHits`, `rt_UpdateContinuousStates`, `rt_UpdateDiscreteEvents`, `rt_UpdatedDiscreteTaskTime`, and `rt_InitTimingEngine` have changed. Change their names to use the prefix `rt_Sim` instead of `rt_` and then change the arguments you pass in to them. See the Release 14 version of `grt_main.c` for the list of arguments passed in to each function.
- Modify all macros that refer to the SimStruct to now refer to the `rtModel`. SimStruct macros begin with the prefix `ss`, whereas `rtModel` macros begin with the prefix `rtm`. For example, change `ssGetErrorStatus` to `rtmGetErrorStatus`.

Generating GRT Wrapper Code from the ERT target

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

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

See the “Code Generation Options and Optimizations” chapter in the Real-Time Workshop Embedded Coder documentation for detailed information on the **GRT compatible call interface** option.

Building Subsystems and Working with Referenced Models

Nonvirtual Subsystem Code Generation (p. 4-2)

Discusses ways to generate separate code modules from nonvirtual subsystems

Generating Code and Executables from Subsystems (p. 4-15)

Describes how to generate and build a stand-alone executable from a subsystem

Generating Code from Models Containing Model Blocks (p. 4-18)

Explains how referenced models can be incorporated into generated code

Nonvirtual Subsystem Code Generation

Real-Time Workshop allows you to control how code is generated for any nonvirtual subsystem. The categories of nonvirtual subsystems are:

- *Conditionally executed* subsystems: execution depends upon a control signal or control block. These include triggered subsystems, enabled subsystems, action and iterator subsystems, subsystems that are both triggered and enabled, and function call subsystems. See the Using Simulink documentation for information on conditionally executed subsystems.
- *Atomic* subsystems: Any virtual subsystem can be declared atomic (and therefore nonvirtual) via the **Treat as atomic unit** option in the **Block Parameters** dialog.

See the Using Simulink documentation, and run the `sl_subsys_semantics` demo for further information on nonvirtual subsystems and atomic subsystems.

You can control the code generated from nonvirtual subsystems as follows:

- You can instruct Real-Time Workshop to generate separate functions, within separate code files if desired, for selected nonvirtual systems. You can control both the names of the functions and of the code files generated from nonvirtual subsystems.
- You can cause multiple instances of a subsystem to generate *reusable* code, that is, as a single reentrant function, instead of replicating the code for each instance of a subsystem or each time it is called.
- You can generate inlined code from selected nonvirtual subsystems within your model. When you inline a nonvirtual subsystem, a separate function call is not generated for the subsystem.

Nonvirtual Subsystem Code Generation Options

For any nonvirtual subsystem, you can choose the following code generation options from the **RTW system code** pop-up menu in the subsystem **Block parameters** dialog:

- **Auto**: This is the default option, and provides the greatest flexibility in most situations. See “Auto Option” below.

- **Inline:** This option explicitly directs Real-Time Workshop to inline the subsystem unconditionally.
- **Function:** This option explicitly directs Real-Time Workshop to generate a separate function with no arguments, and (optionally), place the subsystem in a separate file. You can name the generated function and file. As functions created with this option rely on global data, they are not reentrant.
- **Reusable function:** Generates a function with arguments that allows the subsystem's code to be shared by other instances of it in the model. To enable sharing, Real-Time Workshop must be able to determine (via checksums) that subsystems are identical. The generated function will have arguments for block inputs and outputs, continuous states, parameters, etc.

The sections below further discuss the Auto, Inline, Function, and Reusable function options.

Auto Option

The Auto option is the default, and is generally appropriate. Auto causes Real-Time Workshop to inline the subsystem when there is only one instance of it in the model. When multiple instances of a subsystem exist, the Auto option results in a single copy of the function whenever possible (as a reusable function). Otherwise, the result is as though you selected **Inline** (except for function call subsystems with multiple callers, which is handled as if you specified **Function**). Choose **Inline** to always inline subsystem code, or **Function** when you specifically want to generate a separate function without arguments for each instance, optionally in a separate file.

Note 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**. It is best to use one or the other, as using both creates two reusable functions, one for each designation. The outcomes of these choices differ only when reuse is not possible.

To use the Auto option,

- 1 Select the subsystem block. Then select **Subsystem parameters** from the Simulink **Edit** menu. The **Block Parameters** dialog opens, as shown below.

Alternatively, you can open the **Block Parameters** dialog by

- **Shift**+double-clicking the subsystem block
 - Right-clicking the subsystem block and selecting **Block parameters** from the pop-up menu
- 2 If the subsystem is virtual, select **Treat as atomic unit** as shown in the dialog box below. This makes the subsystem nonvirtual, and the **RTW system code** option becomes enabled.

If the system is already nonvirtual, the **RTW system code** option is already enabled.

- 3 Select **Auto** from the **RTW system code** pop-up menu as shown below.
- 4 Click **Apply** and close the dialog.

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

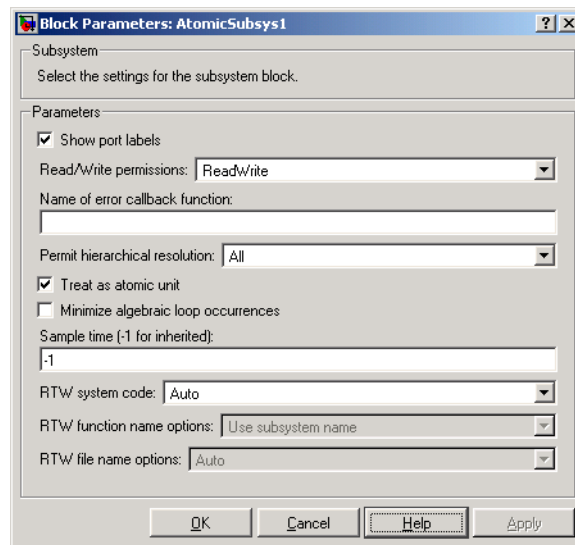


Figure 4-1: Nonvirtual Subsystem Code Generation with Auto Option Selected

Auto Optimization for Special Cases. Rather than reverting to `Inline`, the `Auto` option can optimize code in special situations in which identical subsystems contain other identical subsystems, by both reusing and inlining generated code. Suppose a model, such as schematized in Figure 4-2, contains identical subsystems A1 and A2. A1 contains subsystem B1, and A2 contains subsystem B2, which are themselves identical. In such cases, the `Auto` option causes one function to be generated which is called for both A1 and A2, and this function contains one piece of inlined code to execute B1 and B2, ensuring that the resulting code will run as efficiently as possible.

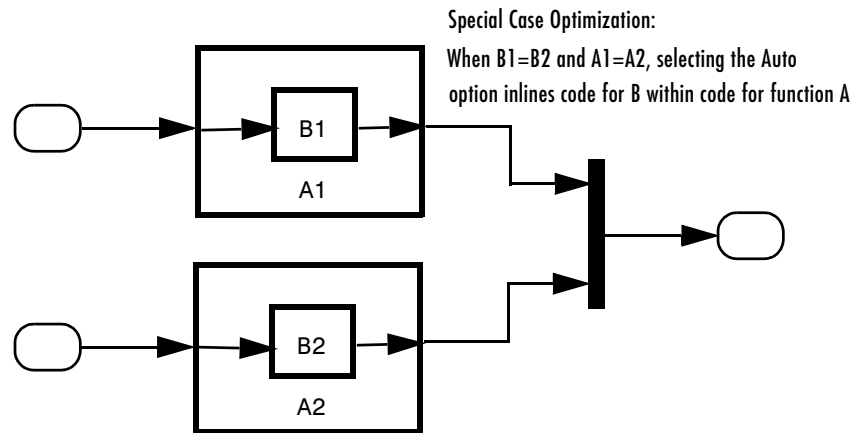


Figure 4-2: Reuse of Identical Nested Subsystems with the Auto Option

Inline Option

As noted above, you can choose to inline subsystem code when the subsystem is nonvirtual (virtual subsystems are always inlined).

Exceptions to Inlining. Note that there are certain cases in which Real-Time Workshop does not inline a nonvirtual subsystem, even though the **Inline** option is selected. These cases are

- 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 such

calls via function pointers; therefore the function-call subsystem must generate a function with all arguments present.

- In a feedback loop involving function-call subsystems, Real-Time Workshop forces one of the subsystems to be generated as a function instead of inlining it. Real-Time Workshop selects the subsystem to be generated as a function based on the order in which the subsystems are sorted internally.
- 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. This might be the case when user-defined Asynchronous Interrupt blocks or Task Synchronization blocks are required. Such blocks must be generated as functions. The VxWorks Asynchronous Interrupt and Task Synchronization blocks, shipped with Real-Time Workshop, use the `SS_OPTION_FORCE_NONINLINED_FCNCALL` option.

To generate inlined subsystem code,

- 1 Select the subsystem block. Then select **Subsystem parameters** from the Simulink **Edit** menu. The **Block Parameters** dialog opens, as shown in Figure 4-3.

Alternatively, you can open the **Block Parameters** dialog by

- **Shift**+double-clicking the subsystem block
 - Right-clicking n the subsystem block and selecting **Block parameters** from the pop-up menu
- 2 If the subsystem is virtual, select **Treat as atomic unit** as shown in Figure 4-3. This makes the subsystem atomic, and the **RTW system code** pop-up menu becomes enabled.

If the system is already nonvirtual, the **RTW system code** menu is already enabled.

- 3 Select **Inline** from the **RTW system code** menu as shown in Figure 4-3.
- 4 Click **Apply** and close the dialog.

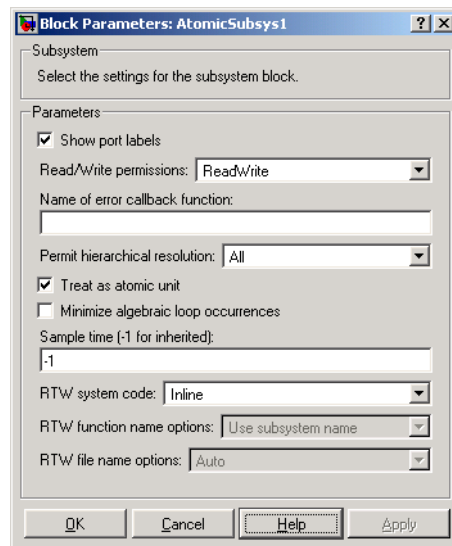


Figure 4-3: Inlined Code Generation for a Nonvirtual Subsystem

When you generate code from your model, Real-Time Workshop writes inline code within *model.c* (or in its parent system’s source file) to perform subsystem computations. You can identify this code by system/block identification tags, such as the following.

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

See “Tracing Generated Code Back to Your Simulink Model” on page 2-45 for further information on system/block identification tags.

Function Option

Choosing the function option or Reusable function lets you direct Real-Time Workshop to generate a separate function and optionally a separate file for the subsystem. When you select the Function option, two additional options are enabled:

- The **RTW function name options** menu lets you control the naming of the generated function.

- The **RTW file name options** menu lets you control the naming of the generated file (if a separate file is generated and you select the User specified option).

Figure 4-4 shows the **Block Parameters** dialog with the Function option selected.

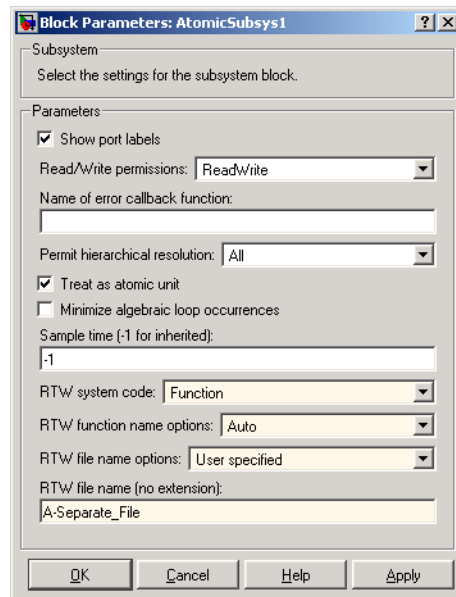


Figure 4-4: Subsystem Function Code Generation with Separate User-Defined File Name

RTW Function Name Options Menu. This menu offers the following choices, but note that the resulting identifiers are also affected by which General code appearance options are in effect for the model:

- Auto: By default, Real-Time Workshop assigns a unique function name using the default naming convention: `model_subsystem()`, where *subsystem* is the name of the subsystem (or that of an identical one when code is being reused).
- Use subsystem name: Real-Time Workshop uses the subsystem name as the function name.

Note When a subsystem is a library block, the `Use subsystem name` option causes its function identifier (and filename, see below) to be that of the library block, regardless of the names used for that subsystem in the model.

- `User specified`: When this option is selected, the **RTW function name** field is enabled. Enter any legal C function name (which must be unique).

RTW File Name Options Menu. This menu offers the following choices:

- `Use subsystem name`: Real-Time Workshop generates a separate file, using the subsystem (or library block) name as the filename.
-

Note When a subsystem's **RTW file name options** is set to **Use subsystem name**, the subsystem file name is mangled if the model contains model blocks, or if a model reference target is being generated for the model. In these situations, the file name for the subsystem consists of the subsystem name prefixed by the model name.

- `Use function name`: Real-Time Workshop generates a separate file, using the function name (as specified by the **RTW function name options**) as the filename.
 - `User specified`: When this option is selected, the **RTW file name (no extension)** text entry field is enabled. Real-Time Workshop generates a separate file, using the name you enter as the filename. Enter any filename desired, but do not include the `.c` (or any other) extension. This filename need not be unique.
-

Note While a subsystem source filename need not be unique, you must avoid giving nonunique names that result in cyclic dependencies (e.g., `sys_a.h` includes `sys_b.h`, `sys_b.h` includes `sys_c.h`, and `sys_c.h` includes `sys_a.h`).

- `Auto`: Real-Time Workshop does *not* generate a separate file for the subsystem. Code generated from the subsystem is generated within the code module generated from the subsystem's parent system. If the subsystem's

parent is the model itself, code generated from the subsystem is generated within *model.c*.

To generate both a separate subsystem function and a separate file,

- 1 Select the subsystem block. Then select **Subsystem parameters** from the Simulink **Edit** menu, to open the **Block Parameters** dialog.

Alternatively, you can open the **Block Parameters** dialog by

- **Shift**+double-clicking the subsystem block
 - Right-clicking the subsystem block and selecting **Block parameters** from the pop-up menu.
- 2 If the subsystem is virtual, select **Treat as atomic unit**. The **RTW system code** menu becomes enabled.

If the system is already nonvirtual, the **RTW system code** menu is already enabled.

- 3 Select **Function** from the **RTW system code** menu as shown in Figure 4-4.
- 4 Set the function name, using the **RTW function name** options described in “RTW Function Name Options Menu” on page 4-8.
- 5 Set the filename, using any **RTW file name** option other than Auto (options are described in “RTW File Name Options Menu” on page 4-9).

Figure 4-4 shows the use of the `UserSpecified` filename option.

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

Reusable Function Option

The difference between functions and reusable functions is that the latter have data passed to them as arguments (enabling them to be reentrant), while the former communicate via global data. Choosing the `Reusable` function option directs Real-Time Workshop to generate a single function (optionally in a separate file) for the subsystem, and to call that code for each identical subsystem in the model, if possible.

Note The `Reusable` function option yields code that is called from multiple sites (hence reused) only when the `Auto` option would also do so. The difference between these options' behavior is that when reuse is not possible, selecting `Auto` yields inlined code (or if circumstances prohibit inlining, creates a function without arguments), while choosing `Reusable` function yields a separate function (with arguments) that is called from only one site.

Limitations on Code Reuse. Subsystems that are superficially identical still might not generate reusable code. Specifically, Real-Time Workshop cannot reuse subsystems having any of the following characteristics:

- Input signals with differing sample times
- Input signals with differing dimensions
- Input signals with differing data type or complexity
- Subsystem masks designating different run-time parameters
- Subsystems containing identical blocks with different names
- Subsystems containing identical blocks with different parameter settings

Some of these situations can arise even when subsystems are copied and pasted within or between models or are manually constructed to be identical. If Real-Time workshop determines that code for a subsystem cannot be reused, it outputs the subsystem as a function with arguments when `Reusable` function is selected, but the function is not reused. If you prefer that subsystem code be inlined in such circumstances rather than deployed as functions, you should choose `Auto` for the **RTW system code** option.

The presence of certain blocks in a subsystem can also prevent its code from being reused. These are

- Scope blocks (with data logging enabled)
- S-function blocks that fail to meet certain criteria
- To File blocks
- To Workspace blocks

Generating Reusable Code from Stateflow Charts. You can generate reusable code from a Stateflow chart, or from a subsystem containing a Stateflow chart, *except* in the following cases:

- The Stateflow chart contains exported graphical functions .
- The Stateflow model contains machine parented events.

Generating Reusable Code for Subsystems Containing S-Function Blocks. Regarding S-function blocks, there are several requirements that need to be met in order for subsystems containing them to be reused. See “Creating Code-Reuse-Compatible S-Functions” in the Simulink documentation.

When you select the Reusable function option, two additional options are enabled. See the explanation of “Function Option” on page 4-7 for descriptions of these options and fields. If you enter names in these fields, you must specify exactly the same function name and filename for each instance of identical subsystems for Real-Time Workshop to be able to reuse the subsystem code.

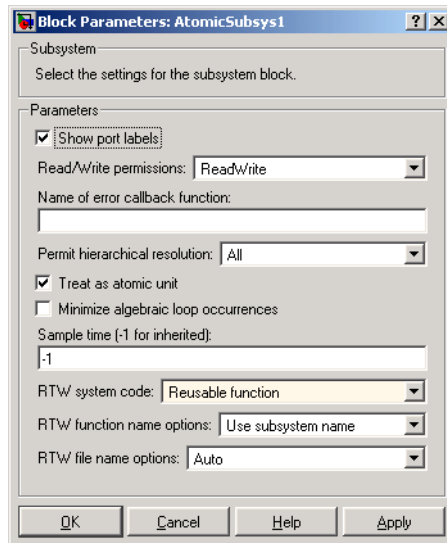


Figure 4-5: Subsystem Reusable Function Code Generation Option

To request that Real-Time Workshop generate reusable subsystem code,

- 1 Select the subsystem block. Then select **Subsystem parameters** from the Simulink **Edit** menu. The **Block Parameters** dialog opens, as shown in Figure 4-3.

Alternatively, you can open the **Block Parameters** dialog by:

- **Shift**+double-clicking the subsystem block
 - Right-clicking the subsystem block and selecting **Block parameters** from the pop-up menu.
- 2 If the subsystem is virtual, select **Treat as atomic unit**. The **RTW system code** menu becomes enabled.

If the system is already nonvirtual, the **RTW system code** menu is already enabled.

- 3 Select **Reusable** function from the **RTW system code** menu as shown in Figure 4-5.
- 4 If you want to give the function a specific name, set the function name, using the **RTW function name** options described in “RTW Function Name Options Menu” on page 4-8.

If you do not choose the **RTW function name** Auto option, and want code to be reused, you must assign exactly the same function name to all other subsystem blocks that you want to share this code.

- 5 If you want to direct the generated code to a specific file, set the filename using any **RTW file name** option other than Auto (options are described in “RTW File Name Options Menu” on page 4-9).

In order for code to be reused, you must repeat this step for all other subsystem blocks that you want to share this code, using the same filename.

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

Generating Reusable Code for Subsystems Containing Model Blocks. You can generate code for subsystems that contain model blocks in the same manner as described above. However, note that the following restrictions apply to such builds:

- You need a license for Real-Time Workshop Embedded Coder in order to build subsystems that contain Model blocks.
- ERT S-functions do not support subsystems that contain a continuous sample time.
- The Real-Time Workshop S-function target is not supported.
- The Tunable parameters table (set via the **Model Parameter Configuration** dialog) 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/or S-function.

Modularity of Subsystem Code

Note that code generated from nonvirtual subsystems, when written to separate files, is not completely independent of the generating model. For example, subsystem code may reference global data structures of the model. Each subsystem code file contains appropriate include directives and comments explaining the dependencies. Real-Time Workshop checks for cyclic file dependencies and warns about them at build time. For descriptions of how generated code is packaged, see “Generated Source Files and File Dependencies” on page 2-60.

Code Reuse Diagnostics

HTML code generation reports (see “Generate HTML Report” on page 2-5) contain a *Subsystems* link in their *Contents* section to a table that summarizes how nonvirtual subsystems were converted to generated code. The *Subsystems* section contains diagnostic information that helps to explain why the contents of some subsystems were not generated as reusable code. In addition to diagnosing exceptions, the HTML report’s *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 atomicdemo demo model.

Generating Code and Executables from Subsystems

Real-Time Workshop can generate code and build an executable from any subsystem within a model. The code generation and build process uses the code generation and build parameters of the root model.

To generate code and build an executable from a subsystem,

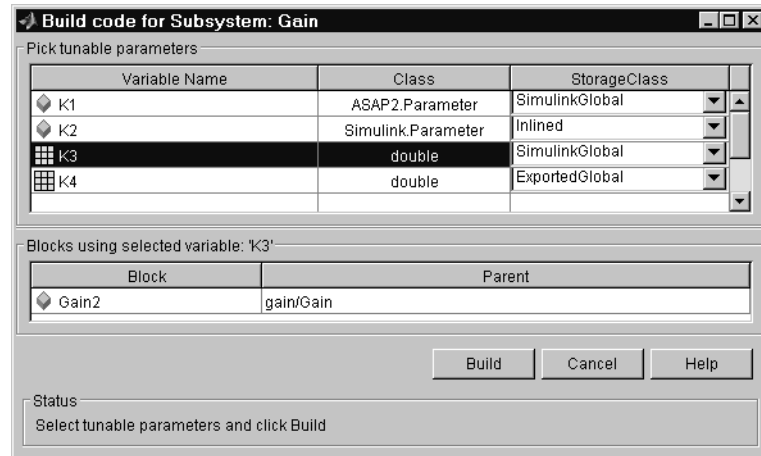
- 1 Set up the desired code generation and build parameters in the **Configuration parameters** dialog, just as you would for code generation from a model.
- 2 Select the desired subsystem block.
- 3 Right-click the subsystem block and select **Build Subsystem** from the **Real-Time Workshop** submenu of the subsystem block's context menu.

Alternatively, you can select **Build Subsystem** from the **Real-Time Workshop** submenu of the **Tools** menu. This menu item is enabled when a subsystem is selected in the current model.

Note If the model is operating in external mode when you select **Build Subsystem**, Real-Time Workshop automatically turns off external mode for the duration of the build, then restores external mode upon its completion.

- 4 The **Build Subsystem** window opens. This 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 all the blocks that reference the parameter and the parent system of each such block.

The **StorageClass** column contains a popup menu for each row. The menu lets you set the storage class of any parameter or inline the parameter. To inline a parameter, select the **Inline** option from the menu. To declare a parameter to be tunable, set the storage class to any value other than **Inline**.



In the illustration above, the parameter K2 is inlined, while the other parameters are tunable and have various storage classes.

See “Parameters: Storage, Interfacing, and Tuning” on page 5-2 and “Simulink Data Objects and Code Generation” on page 5-33 for further information on tunable and inlined parameters and storage classes.

- 5 After selecting tunable parameters, click the **Build** button. This initiates the code generation and build process.
- 6 The build process displays status messages in the MATLAB Command Window. When the build completes, the generated executable is in your working directory. The name of the generated executable is *subsystem.exe* (PC) or *subsystem* (UNIX), where *subsystem* is the name of the source subsystem block.

The generated code is in a build subdirectory, named *subsystem_target_rtw*, where *subsystem* is the name of the source subsystem block and *target* is the name of the target configuration.

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 Subsystem** behaves identically to **Generate S-function**. (See “Automated S-Function Generation” on page 11-12.)

Generating Code from Models Containing Model Blocks

The *Model block* from the Simulink Ports & Subsystems library allows one model to include another model as if it were a block. This feature, called *model reference*, works by generating code for included models that the parent model executes from a binary library file. Model reference works on all UNIX and Linux platforms (using the gcc compiler), and on Windows PC platforms (using the lcc and the Visual C++ compilers). This section covers the following topics:

“About Model Reference” on page 4-18	Explains model reference terms and concepts, with pointers to additional information
“Using Referenced Models” on page 4-21	Describes options, limitations, and compatibility issues when generating and interfacing to code from model blocks
“Project Directory Structure for Model Reference Targets” on page 4-27	Describes how model reference code is organized, including directory structure, and options for utilities and makefiles
“Supporting the Shared Utilities Directory in the Build Process” on page 4-33	Describes how to include code from project directories in stand-alone models with or without model references
“Inherited Sample Time for Referenced Models” on page 4-35	Explains how referenced models can inherit sample time, how to modify S-functions to permit this, and certain limitations
“Reusing Code for Subsystems in Referenced Models” on page 4-40	Explains why subsystem code might not be reused in referenced models, and how to modify a model to enable code reuse
“Making Custom Targets Model Reference Compatible” on page 4-42	Provides guidelines for including code from referenced models in your own custom targets

About Model Reference

We call models that include Model blocks *top models*. Model referencing uses *incremental loading*; when a top model is opened, any models it references are not loaded into memory until they are needed or the user opens them.

Note In order to take advantage of incremental model loading, models called from Model blocks must be saved at least once with the current version of Simulink (Release 14).

When running simulations, models include other models by generating code for them in a project directory (see below) and creating a static library file called a *simulation target* (sometimes referred to as a SIM target, which is not the same as the *RSim* rapid simulation target). When Real-Time Workshop generates code for referenced models, it follows a parallel process to create whatever type of target (e.g., GRT) you have specified; these are generically referred to as *RTW targets*. The generated code is also stored in the project directory, although generated code for parent models is stored (as previously) in a build directory at the same level as the model reference project directory.

In addition to incremental loading, the model referencing mechanism employs *incremental code generation*. This is accomplished by comparing the date, and optionally, the structure of model files of referenced models with those for their generated code to determine whether it is necessary to regenerate model reference targets. You can also force or prevent code generation via a diagnostic setting, **Rebuild options**, in the **Model Referencing** configuration dialog.

Model Reference Demos and Tutorial

You can learn more about how Model blocks work and generate code by running demos. For the full demo suite, at the MATLAB prompt type

```
mdlrefdemos
```

The suite contains three separate demos:

- `mdlref_basic` — General demonstration of using Model blocks
- `mdlref_paramargs` — Passing parameters to referenced models
- `mdlref_bus` — Using bus objects to communicate signals to referenced models
- `mdlref_conversion` — Automatically converting atomic subsystems in models to models called with Model blocks

For an example of model reference code generation, including use of the `mdlref_conversion` demo, see “Tutorial 6: Generating Code for a Referenced Model” on page 3-51.

Generating Code for Models with Model Blocks

When a model includes one or more other models, Simulink generates code for the referenced models and uses it to build shared library files for updating the diagram and simulation. This happens automatically, as described below. Code for building Real-Time Workshop applications is generated when you click the **Build** button on the **Real-Time Workshop** configuration dialog. Other ways you can start the code generation process are to select **Tools -> Real-Time Workshop -> Build Model** from the Simulink menu in the model window, and to press **Ctrl+B**.

Note You cannot build models that contain Model blocks using the Real-Time Workshop S-function target. This also means that you cannot build a subsystem module by right-clicking (or via **Tools -> Real-Time Workshop -> Build subsystem**) if the subsystem contains Model blocks. This restriction applies only to GRT S-functions, not to ERT S-functions.

If you only want to generate code for referenced models without generating code for the top model, in the MATLAB Command Window type:

```
slbuild('model', 'ModelReferenceRTWTarget')
```

Code for referenced models is generated into the `slprj` directory.

Note that model reference executables for *simulations* are required when you generate code in Real-Time Workshop. Model reference files for simulation are rebuilt (if necessary) when you run a simulation or update the diagram. Whether these files are rebuilt depends on how your model has changed and on your **Rebuild options** setting on the **Model Reference** pane of the configuration dialog. You can update a model reference simulation target by typing

```
slbuild('model', 'ModelReferenceSimTarget')
```

For further information on building model reference simulation targets, see the Simulink documentation.

Project Directories

When models referenced via Model blocks are built for simulation or Real-Time Workshop code generation, files are placed in a project directory named `s1prj` within the current working directory. The subdirectories in `s1prj` provide separate places for simulation code, Real-Time Workshop code, and other files. For details, see the release note “Project Directory Structure for Model Reference Targets” on page 4-27.

Using Referenced Models

To include one model in another (called the top model), you insert it as a Model block, and configure it using a set of controls in the dialog for the **Model Referencing** configuration component. You can access the controls via Model Explorer or via standalone dialogs. For details on using the **Model Referencing** dialog, see “Model Referencing Pane” in the Simulink documentation.

Parameterizing Referenced Models

In addition to the controls on the **Model Referencing** dialog described above, you can also specify parameters to be passed to a referenced model when it is called. See “Parameterizing Model References” in the Simulink documentation for further information.

MAT-File Logging for Model Reference Targets

Top-level models can perform data logging (writing output, time steps, and states to a MAT-file) whether they reference any models or not. Generated code for referenced models, on the other hand, does not log data to MAT-files regardless of which RTW target is produced. If a referenced model is set up to log data to a MAT-file, Real-Time Workshop silently turns this option off during RTW target code generation, then restores it after the build completes.

Model Reference Diagnostics

The **Model Referencing** pane of the **Diagnostics** portion of the **Configuration Parameters** dialog provides important controls that can help you configure Model blocks. See “Model Reference Diagnostics” in the Simulink documentation for specific details.

Possible Incompatibilities Between Top and Referenced Models

A model and its referenced models can have differences in option settings that conflict when Real-Time Workshop generates code. Some incompatibilities can be ignored or handled as warnings using diagnostic settings. Other incompatibilities always result in errors during code generation, and must be remedied by changing settings in either top models or referenced models. The following table lists Real-Time Workshop **Configuration Parameters** dialog options that can conflict if set in certain ways or if set differently in a referenced model than in a top-level model. Note that some of these conditions only apply to ERT and ERT-derived targets.

Dialog	Option	Remarks
Optimization	Inline parameters	Must be <i>on</i> for top and referenced models
Real-Time Workshop	RTW system target file	Must be the same for top and referenced models
Real-Time Workshop	Generate code only	Must be the same for top and referenced models
Real-Time Workshop	Ignore custom storage classes	(ERT-derived targets only) Must be the same for top and referenced models
Symbols	Maximum identifier length	Can be longer for top model than referenced models
Interface	Target floating-point math environment	Must be the same library for top and referenced models
Interface	Support floating-point numbers	(ERT-derived targets only) If <i>off</i> for top model, must be <i>off</i> for referenced models
Interface	Support complex numbers	(ERT-derived targets only) If <i>off</i> for top model, must also be <i>off</i> for referenced models

Dialog	Option	Remarks
Interface	Support nonfinite numbers	(ERT-derived targets only) If <i>off</i> for top model, must also be <i>off</i> for referenced models
Optimization	Application lifespan (days)	Must be the same for top and referenced models
Interface	Terminate function required	(ERT-derived targets only) Must be the same for top and referenced models
Interface	Suppress error status in real-time model	(ERT-derived targets only) If <i>on</i> for top model, must also be <i>on</i> for referenced models
Interface	Data Exchange Interface: C-API	Signals and Parameters check boxes must be in same states for top and referenced models
Interface	Data Exchange Interface: ASAP2	Must be the same for top and referenced models
Templates	Target operating system	(ERT-derived targets only) Must be the same for top and referenced models
Data Placement	MPT Module Naming	(ERT-derived targets only) Must be the same for top and referenced models
Data Placement	MPT Module Name	(ERT-derived targets only) Must be the same for top and referenced models
Data Placement	MPT Source of initial values	(ERT-derived targets only) Must be the same for top and referenced models

In addition, be aware of the following conditions:

- **Solver** — Only one solver is used for all models.
 - When a referenced model uses a different solver from the top model, its solver setting is ignored and the top model’s solver is used.
 - If the top model uses a fixed-step solver, and a referenced model has any continuous states, issue a diagnostic if solvers differ.
 - If the top model solver is fixed-step and a referenced model contains continuous-time blocks, issue a diagnostic.

Note When you build a Rapid Simulation (RSim) target, all referenced models must use fixed-step solvers. The top model, however, can use a variable-step solver as long as all blocks in the referenced models are discrete.

- **Data Import/Export** — The **Load initial state** option must be *off* when building a target for a referenced model; it can be *on* for the top model.

Note Custom targets should declare themselves to be model reference compliant if they need to support Model blocks. For details on accomplishing this, see “Making Custom Targets Model Reference Compatible” on page 4-42.

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, be aware of how the signal data will be handled, and be careful not to violate any of several restrictions on their use.

A global signal is a signal with a non-Auto storage class:

- ExportedGlobal
- ImportedExtern
- ImportedExternPointer
- Custom

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

- An extern declaration is generated by all models that use any 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 any exported signal, the topmost model that uses the signal is responsible for defining (allocating memory for) the signal.
Therefore if a signal crosses a model's boundary, that model is not responsible for defining the signal. Instead, the parent model will generate the definition.
- All global signals are accessed directly (as global memory). They are not passed as arguments to any of the functions that are generated for any of 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. These include

- Signals with a grouped custom storage class (e.g., Struct) must be declared as Imported.
- Ungrouped signals cannot be exported via a special user-defined file. Such signals are exported via `model.h`.

Storage Classes for Parameters Used with Model Blocks

Note the following limitations regarding handling of parameters for referenced models:

- Inline parameters off is not supported.
- Tunable parameters are not supported for noninlined S-functions.
- Tunable parameters set using the **Model Parameter Configuration** dialog are ignored.

The above rules apply to the built-in storage classes and custom storage classes alike. You should also read “Parameterizing Model References” in the Simulink documentation for further information on how you can control parameter passing to referenced models.

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

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

Some key considerations to remember:

- All tunable parameters are treated as `SimulinkGlobal` for simulation. As a result, all storage classes are implicitly supported.
- 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 non-`Auto` storage class.
 - An extern declaration is generated by all models that use any given parameter.
 - All global parameters are “bubbled up” to the top model.
If a parameter is exported, the top model is responsible for defining (allocating memory for) the parameter (whether it uses the parameter or not).
 - All global parameters are accessed directly (as global memory). They are not passed as arguments to any of the functions that are generated for any of the referenced models.
 - Code for `Auto` parameters that are tunable (`SimulinkGlobal`) in referenced models is generated using unstructured variables (`rtP_xxx`) instead of being written into the `model_P` (formerly `rtP`) structure. This is so that the parameters can be seen by model reference code.
- As in the case of signals, certain custom storage classes for parameters are not currently supported for model reference. These include
 - Parameters with a grouped custom storage class (e.g., `struct`) must be declared as `Imported`.
 - Ungrouped parameters cannot be exported via a special user-defined file; they are exported via `model.h`.
- Parameters used in Model block arguments must be defined in the referenced model’s workspace. See “Parameterizing Model References” in the Simulink documentation for specific details.

Project Directory Structure for Model Reference Targets

Code for models referenced via Model blocks is generated in project directories within the current working directory. The top-level project directory is always named `/slprj`. The next level within `slprj` contains parallel build area subdirectories for simulation targets (`/sim`) and Real-Time Workshop targets (`/target`, where *target* is GRT, ERT, etc.).

The principal project directories/files are

- Model reference SIM (Simulation) target files:

```
slprj/sim/modelName/
```

- MAT-files used during code generation of Model reference SIM target and accelerator:

```
slprj/sim/modelName/tmwinternal
```

- Model reference RTW target files:

```
slprj/target/modelName/
```

- MAT-files used during code generation of Model reference RTW target and stand-alone code generation:

```
slprj/target/modelName/tmwinternal
```

- `slprj` marker file

```
slprj/sl_proj.tmw (marker file)
```

- Utility functions for RTW targets, shared across models:

```
slprj/target/_sharedutils
```

- Utility functions for SIM targets, shared across models:

```
slprj/sim/_sharedutils
```

In the above paths, *modelName* is the name of the model being used as a referenced model, and *target* is the Real-Time Workshop target acronym (e.g., `grt`, `ert`, `rsim`, etc.).

If you are building code for more than one referenced model within the same working directory, model reference files for all such models are added to the

existing slprj directory. For example, below is a directory listing for a project containing two referenced models, that are configured for the ERT target:

```
\slprj
|   sl_proj.tmw
|
+---sim
|   +---arch
|   |   \---tmwinternal
|   |           (project housekeeping, not for user)
|   |
|   +---engine3200cc
|   |   |   engine3200cc_capi.c
|   |   |   engine3200cc_capi.h
|   |   |   engine3200cc_types.h
|   |   |   engine3200cc.c
|   |   |   engine3200cc_private.h
|   |   |   engine3200cc.h
|   |   |   engine3200cc_msf.c
|   |   |   modelsources.txt
|   |   |   rtw_proj.tmw
|   |   |   engine3200cc.mk
|   |   |   engine3200cc.o
|   |   |   engine3200cc_capi.o
|   |   |   engine3200cclib.a
|   |   |
|   |   +---tmwinternal
|   |   |           (project housekeeping, not for user)
|   |   |
|   |   \---tlc
|   |           (code generation support, not for user)
|   |
|   +---transmission
|   |   |   transmission_capi.c
|   |   |   transmission_capi.h
|   |   |   transmission_types.h
|   |   |   transmission.c
|   |   |   transmission_private.h
|   |   |   transmission.h
|   |   |   transmission_msf.c
```



```
| | engine3200cc.o
| | engine3200cc_rtwlib.a
| |
+---tmwinternal
| | (project housekeeping, not for user)
| |
\---tlc
| | (code generation support, not for user)
| |
+---transmission
| | transmission_types.h
| | transmission.c
| | transmission_private.h
| | transmission.h
| | modelsources.txt
| | rtw_proj.tmw
| | transmission.mk
| | transmission.o
| | transmission_rtwlib.a
| |
+---tmwinternal
| | (project housekeeping, not for user)
| |
\---tlc
| | (code generation support, not for user)
| |
\---_sharedutils
| | checksummap.mat
| | FIX2FIX_U4_U16.c
| | FIX2FIX_U4_U16.h
| | FIX2FIX_U12_U16.c
| | FIX2FIX_U12_U16.h
| | rtwtypes.h
| | rtw_shared_utils.h
| | rt_nonfinite.h
| | rt_nonfinite.c
| | FIX2FIX_U12_U16.o
| | FIX2FIX_U4_U16.o
| | rt_nonfinite.o
```

Sharing Utility Code

If no Model blocks are present, any code required for fixed point and other shared utilities is placed in the *model.c* file or in the build directory, not in the `slprj/target/_sharedutils` directory.

To force a build to use the `slprj/target/_sharedutils` directory even when the current model contains no model blocks, set the **Utility function generation** pull-down on the **Real-Time Workshop Interface** pane to Shared location. This places utilities in the `slprj` directory rather than in the build directory.

The other option (Auto) does this only when there are referenced models. That is, if a model does contain Model blocks, then the Auto setting of the **Utility function generation** drop-down uses the shared utilities directory within `slprj`, as Model Reference builds require the use of shared utilities.

Code in files in the `_sharedutils` directory differs depending on what target configurations exist (e.g., whether fixed-point utilities are required). The configuration information for each target is saved in `_sharedutils/checksummap.mat`.

Makefile Requirements

The makefile used by Real-Time Workshop must support compiling and creating libraries, etc., from the locations in which the code is generated. You therefore will need to update your makefile to support the shared utilities location, and additionally the model reference build process. The changes required for a makefile are documented in “Supporting the Shared Utilities Directory in the Build Process” on page 4-33 and “Template Makefile Modifications” on page 4-43.

In terms of makefile support, the **Utility function generation** options have the following requirements:

- Auto
 - Stand-alone model build — All files go to the build directory; no makefile updates needed.
 - Referenced model or top model build — Use shared utilities directory; makefile requires full model reference support.
- Shared location
 - Stand-alone model build — Use shared utility directory; makefile requires shared location support.

- Referenced model or top model build — Use shared utilities directory; makefile requires full model reference support.

Customization Notes

You can use an M-file API for determining whether code is generated using shared utilities:

```
matlabroot/toolbox/rtw/rtw/rtw_gen_shared_utils(modelName)
```

`rtw_gen_shared_utils` returns 1 if the build will be using shared utilities, and 0 otherwise. This indicates whether generated code for utilities is written to the `slprj` shared utilities directory or to the model build directory, respectively. The shared utilities directory is used when the current model is included via a Model block in another model or when generating code when the current model is the top model and itself contains model blocks.

You can call `rtw_gen_shared_utils` only after `PrepareBuildArgs` has been called (that is, you can only invoke it after reaching the `before_tlc` stage when processing `target_make_rtw_hook.m`).

The following TLC variables are also available:

- `GenUtilsSrcInSharedLocation` — 1 if utilities will be shared, 0 otherwise
- `GenUtilsPath` — Full path to the location for utils functions

Header File Changes

For code generation purposes, the inclusion of `tmwtypes.h` and `simstruc_types.h` has been replaced by inclusion of `rtwtypes.h`. For non-ERT targets, `rtwtypes.h` simply includes `tmwtypes.h` and `simstruc_types.h` and is thus equivalent. For ERT targets, `rtwtypes.h` itself provides the necessary defines, enumerations, etc., instead of including `tmwtypes.h` and `simstruc_types.h`. The `rtwtypes.h` generated for ERT is an optimized (reduced) file based on the settings provided with the model being built.

For both ERT and non-ERT targets, the location of the generated `rtwtypes.h` is controlled by whether the build is using the shared utilities location. If shared location is being used, `rtwtypes.h` is generated into `slprj/target/_sharedutils`; the other `rtwtypes` are generated into the standard build directory location (`model_target_rtw`). See “Sharing Utility Code” on page 4-31 for more information on when and how the shared utilities location is used.

Supporting the Shared Utilities Directory in the Build Process

The shared utilities directory (`s1prj/target/_sharedutils`) typically stores generated utility code that is common to a top-level model and the models it references. You can also force the build process to use a shared utilities directory for a standalone model. See “Sharing Utility Code” for details.

If you want your target to support compilation of code generated in the shared utilities directory, several updates to your template makefile (TMF) are required. Note that support for the shared utilities directory is a necessary, but not sufficient, condition for supporting Model Reference builds. See “Making Custom Targets Model Reference Compatible” on page 4-42 to learn about additional updates that are needed 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 make utility. You can find the following updated TMF examples for GNU and Microsoft Visual C make utilities in the GRT and ERT target directories:

- GRT: `matlabroot/rtw/c/grt/`
 - `grt_lcc.tmf`
 - `grt_vc.tmf`
 - `grt_unix.tmf`
- ERT: `matlabroot/rtw/c/ert/`
 - `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.

Modifying Template Makefiles to Support Shared Utilities

Make the following changes to your TMF to support the shared utilities directory:

- 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 directory 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 directories for shared source files and the library compiled from the source files. In the current release, all TMFs actually 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 all 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 UNIX based.

```
$(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 any explicit reference to `rt_nonfinite.c` from your TMF. For example, change

```
ADD_SRCS = $(RTWLOG) rt_nonfinite.c
```

to

```
ADD_SRCS = $(RTWLOG)
```

Inherited Sample Time for Referenced Models

It is sometimes desirable for a Model block to inherit a sample time. Without this ability, a Model block cannot be placed in a triggered subsystem (or function call, or iterator). Additionally, allowing a Model block to inherit sample time in a variety of contexts maximizes its reuse potential. For

example, a model might fix the data types and dimensions of all its input and output signals, but could be reused with different sample times, e.g., discrete at 0.1, discrete at 0.2, triggered, etc. If the blocks it contains meet certain requirements, there is no reason why such a model cannot inherit any discrete sample time when used as a Model block.

Enabling Model Blocks to Inherit Sample Time

If you want a Model block to be used in a model where it can inherit a sample time, you must constrain the solver declared for that model. On the **Solver** configuration dialog pane, set solver **Type** to **Fixed-step** and **Periodic sample time constraint** to **Ensure sample time independent**. When Simulink generates Model block code for that model, it halt with an error if this model is unable to inherit sample times.

A model is only allowed to inherit a sample time if and only if *all* the following conditions are true:

- None of its blocks specifies sample times (other than inherited and constant).
- No fixed step size has been specified for the solver by the user.
- After sample time propagation, there is only one sample time in the model (not counting constant sample time).
- No S-functions make use of their specific sample time internally.

Determining Whether an S-function Makes Internal Use of Sample Time

In order for Simulink to accurately determine whether a model can inherit a sample time when the model is used in a Model block, its S-functions need to specify what uses they make of their sample times. Use the `ssSetModelReferenceSampleTimeInheritanceRule` macro, which you can call at any point from `mdlInitializeSizes` to `mdlSetWorkWidths` to provide this information. The following guidelines describe how to use this macro:

- 1 Check for use of any of the following macros anywhere in the S-function code:

In your S-function C code:

- `ssGetSampleTime`
- `ssGetInputPortSampleTime`
- `ssGetOutputPortSampleTime`

- `ssGetInputPortOffsetTime`
- `ssGetOutputPortOffsetTime`
- `ssGetSampleTimePtr`
- `ssGetInputPortSampleTimeIndex`
- `ssGetOutputPortSampleTimeIndex`
- `ssGetSampleTimeTaskID`
- `ssGetSampleTimeTaskIDPtr`

In your S-function TLC code:

- `LibBlockSampleTime`
- `CompiledModel.SampleTime`
- `LibBlockInputSignalSampleTime`
- `LibBlockInputSignalOffsetTime`
- `LibBlockOutputSignalSampleTime`
- `LibBlockOutputSignalOffsetTime`

- 2** If none of the above is present, your S-function does not preclude the model from inheriting a sample time. Register this fact as follows:

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

- 3** If any of the above macros are used, but only for any of the following,
- Throwing errors if sample time is inherited, continuous, or constant
 - Checking `ssIsSampleHit`
 - Checking whether sample time is inherited in either `mdlSetInputPortSampleTime` or `mdlSetOutputPortSampleTime` before setting

then set

```
ssSetModelReferenceSampleTimeInheritanceRule...
(S,USE_DEFAULT_FOR_DISCRETE_INHERITANCE)
```

- 4** If the S-function uses its sample time for computing parameters, outputs, etc., then set

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

Note If an S-function does not set this flag, then by default Simulink assumes that the S-function does not preclude the model containing it from inheriting a sample time. However, Simulink does issue a message warning you that unset S-functions are present.

You can use settings on the **Solver** tab of the **Diagnostics** pane in the configuration dialog or Model Explorer to tell Simulink how to behave when it encounters S-functions that have unspecified sample time inheritance rules. You can toggle the **Unspecified inheritability of sample time** diagnostic to none, warning, or error. The default is warning.

Examples

You can preclude inheriting sample time or not 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 demo `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 < NOOUTPUTS; i++) {
            if (ssIsSampleHit(S,
                ssGetOutputPortSampleTimeIndex(S,i), tid)) {
                real_T *y = ssGetOutputPortRealSignal(S,i);
                *y = signal;
            }
        }
    }
} /* end mdlOutputs */

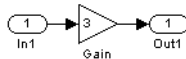
```

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

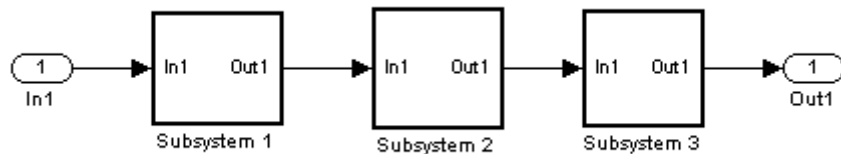
```
ssSetModelReferenceSampleTimeInheritanceRule
(S, USE_DEFAULT_FOR_DISCRETE_INHERITANCE);
```

Reusing Code for Subsystems in Referenced Models

Real-Time Workshop can generate code for a model's atomic subsystems as reusable functions, under your control. When such a model is referenced by a top model via a Model block, function signatures for identical atomic subsystems it contains can vary, depending on whether they are connected to a root inport, an outport, or neither. This means that code for certain atomic subsystems cannot be reused in a model reference context the same way it was in the stand-alone model. For example, here is a subsystem marked **Treat as an atomic unit**, and which has its **RTW system code** set to reusable function:



The model shown below includes three of these subsystems, code for which can be successfully reused in the stand-alone model, but cannot be reused in a model reference context:



When generated from a top (root level) model (with inline parameters on), code for the three subsystems is collapsed into a single function:

```
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 `s1prj` project directory) the subsystems have three different function signatures:

```

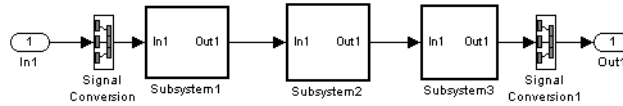
/* Output and update for atomic system: '<Root>/Subsystem1' */
void reuse_subsys1_Subsystem1(const real_T *rtu_0,
rtB_reuse_subsys1_Subsystem1
*localB)
{
    /* Gain: '<S1>/Gain' */
    localB->Gain_w = (*rtu_0) * 3.0;
}

/* 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—and therefore assure code reuse—is to insert Signal Conversion blocks. Place one between the Inport and Subsystem1 and another between Subsystem3 and the Output of the referenced model, as follows:



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 if you select the **Pass scalar root inputs by value** check box at the bottom of the **Model Referencing** configuration dialog. Note that when you do this, you still need to insert a Signal Conversion block before the Outport.

Making Custom Targets Model Reference Compatible

Models that employ model reference might require special treatment when generating code for custom targets. The following sections describe how to adapt your custom target for code generation compatibility with the model reference features introduced in Release 14. Most of the guidelines below concern required modifications to your system target file (STF) and template makefile (TMF).

General Considerations

- A model reference compatible target must be derived from the ERT or GRT targets.
- When generating code from a model that references another model, both the top-level model and the referenced models must be configured for the same code generation target.

- Note that the **External mode** option is not supported in model reference Real-Time Workshop target builds. If the user has selected this option, it is ignored during code generation.
- To support model reference builds, your TMF must support use of the shared utilities directory, as described in “Supporting the Shared Utilities Directory in the Build Process” on page 4-33.

System Target File Modifications

- Your STF must implement a `SelectCallback` function (see “SelectCallback Function for System Target Files” on page 4-46). Your `SelectCallback` function must declare model reference compatibility by setting the `ModelReferenceCompliant` flag.

The callback is executed if the function is installed in the `SelectCallback` field of the `rtwgensettings` structure in your STF. The following code installs the `SelectCallback` function:

```
rtwgensettings.SelectCallback =
    ['custom_open_callback_handler(hDlg, hSrc)'];
```

Your callback should set the `ModelReferenceCompliant` flag as follows.

```
slConfigUISetVal(hDlg, hSrc, 'ModelReferenceCompliant', 'on');
```

See “Compatibility Issues for `rtwoptions` Callbacks” in the Real-Time Workshop Embedded Coder 4.0 release notes for details on the callback API, including `slConfigUISetVal`.

Template Makefile Modifications

In addition to the TMF modifications described in “Supporting the Shared Utilities Directory in the Build Process” on page 4-33, you must modify your TMF variables and rules as described below.

- 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_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 excerpts show how makefile tokens are expanded for a referenced model, and for the top-level model that references it.

```
MODELREFS                  =  
MODELLIB                   = engine3200cc_rtwlib.a  
MODELREF_LINK_LIBS        =  
MODELREF_INC_PATH         =  
RELATIVE_PATH_TO_ANCHOR   = ../../..  
MODELREF_TARGET_TYPE      = RTW
```

```
Example of how tokens are expanded for the top-level model  
MODELREFS                  = engine3200cc transmission  
MODELLIB                   = archlib.a  
MODELREF_LINK_LIBS        = engine3200cc_rtwlib.a transmission_rtwlib.a  
MODELREF_INC_PATH         = -I../slprj/ert/engine3200cc  
-I../slprj/ert/transmission  
RELATIVE_PATH_TO_ANCHOR   = ..  
MODELREF_TARGET_TYPE      = NONE
```

The `MODELREFS` token for the top-level model expands to a list of referenced model names.

The `MODELLIB` token expands to the name of the library generated for the model.

The `MODELREF_LINK_LIBS` token for the top-level model expands to a list of referenced model libraries that the top-level model links against.

The `MODELREF_LINK_LIBS` token for the top-level model expands to the include path to the referenced models.

The `RELATIVE_PATH_TO_ANCHOR` token expands to the relative path, from the location of the generated makefile, to the MATLAB working directory (pwd).

The `MODELREF_TARGET_TYPE` token signifies the type of target being built. Possible values are

- `NONE`: Standalone model or top-level model referencing other models
- `RTW`: Model reference Real-Time Workshop target build

- SIM: Model reference simulation target build

- 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. Further modules will then be appended conditionally, as described in step 4 below. 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 any variables that might have existed for defining the final target. For example, remove

```
PROGRAM = ../$(MODEL)
```

and replace it with

```
ifeq ($(MODELREF_TARGET_TYPE), NONE)
  # Top-level 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 final target of makefile (replace any existing final target rule). For example,

```
ifeq ($(MODELREF_TARGET_TYPE),NONE)
# Top-level model for RTW
$(PRODUCT) : $(OBJS) $(SHARED_OBJS) $(MODELREF_LINK_LIBS) $(LIBS)
              $(BIN_SETTING) $(LINK_OBJS) $(SHARED_OBJS)
              $(MODELREF_LINK_LIBS) $(LIBS)
              @echo "### Created $(BUILD_PRODUCT_TYPE): $@"
else
# sub-model for RTW
$(PRODUCT) : $(OBJS) $(SHARED_OBJS)
              @rm -f $(MODELLIB)
              $(AR) ruv $(MODELLIB) $(LINK_OBJS)
              @echo "### $(MODELLIB) Created"
              @echo "### Created $(BUILD_PRODUCT_TYPE): $@"
endif
```

- 6** Create a rule to allow submodels to compile files that reside in the MATLAB working directory (pwd).

```
%.o : $(RELATIVE_PATH_TO_ANCHOR)/%.c
      $(CC) -c $(CFLAGS) $<
```

SelectCallback Function for System Target Files

The Release 14 API for system target file callbacks provides a new function for use in system target files. `SelectCallback` is associated with the target rather than with any of its individual options. If a `SelectCallback` function is implemented for the target, it is triggered once when the user selects the target via the System Target File browser.

To implement this callback, use the `SelectCallback` field of the `rtwgensettings` structure. The following code installs a `SelectCallback` function:

```
rtwgensettings.SelectCallback =
['custom_open_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 system target file callback functions. They should be passed in without alteration, as in this example:

```
slConfigUISetVal(hDlg, hSrc, 'ModelReferenceCompliant', 'on');
```

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 “Making Custom Targets Model Reference Compatible” on page 4-42 for an example.

Working with Data Structures

Parameters: Storage, Interfacing, and Tuning (p. 5-2)	How to generate storage declarations for communicating model parameters to and from user-written code
Signal Storage, Optimization, and Interfacing (p. 5-19)	How signal storage optimizations work, and how to generate storage declarations for communicating model signals to and from user-written code
Simulink Data Objects and Code Generation (p. 5-33)	How to represent and store signals and parameters in Simulink data objects, and how code is generated from these objects
Block States: Storing and Interfacing (p. 5-47)	How to generate storage declarations for communicating discrete block states to and from user-written code
Storage Classes for Data Store Memory Blocks (p. 5-56)	How to control data structures that define and initialize named shared memory regions, used by the Data Store Read and Data Store Write blocks

Parameters: Storage, Interfacing, and Tuning

Simulink external mode (see Chapter 6, “External Mode”) offers a quick and easy way to monitor signals and modify parameter values while generated model code executes. However, external mode might not be appropriate for your application in some cases. The S-function target does not support external mode, for example. For other targets, you might want your existing code to access parameters and signals of a model directly, rather than using the external mode communications mechanism.

This section discusses how Real-Time Workshop generates parameter storage declarations, and how you can generate the storage declarations you need to interface block parameters to your code. For guidance on implementing a parameter tuning interface using a C-API, see “C-API for Interfacing with Signals and Parameters” on page 17-2.

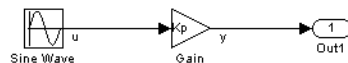
Storage of Nontunable Parameters

By default, block parameters are not tunable in the generated code. In the default case, Real-Time Workshop has control of parameter storage declarations and the symbolic naming of parameters in the generated code.

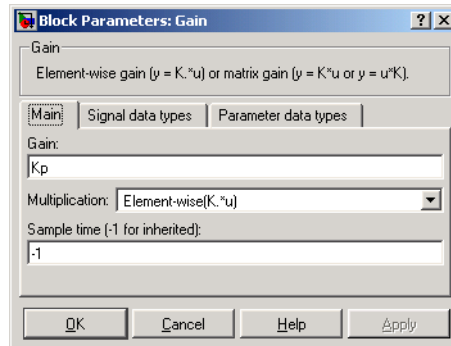
Nontunable parameters are stored as fields within *model_P* (formerly *rtP*), a model-specific global parameter data structure. Real-Time Workshop initializes each field of *model_P* to the value of the corresponding block parameter at code generation time.

When the **Inline parameters** option is on, block parameters are evaluated at code generation time, and their values appear as constants in the generated code. (A vector parameter, however, can be represented as an array of constants within the Constant Parameter Structure.)

As an example of nontunable parameter storage, consider this model.



The workspace variable `Kp` sets the gain of the Gain1 block.



Assume that `Kp` is nontunable and has a value of 5.0. The table below shows the variable declarations and the code generated for `Kp` in the noninlined and inlined cases.

Notice that the generated code does not preserve the symbolic name `Kp`. The noninlined code represents the gain of the Gain1 block as `model_P.Gain1_Gain`.

Nontunable Parameter Storage Declarations and Code

Inline Parameters	Generated Variable Declaration and Code
Off	<pre> struct _Parameters_non_tunable_sin { real_T SineWave_Amp; real_T SineWave_Bias; real_T SineWave_Freq; real_T SineWave_Phase; real_T Gain_Gain; }; . . . Parameters_non_tunable_sin non_tunable_sin_P = { 1.0 , /* SineWave_Amp : '<Root>/Sine Wave' */ 0.0 , /* SineWave_Bias : '<Root>/Sine Wave' */ 1.0 , /* SineWave_Freq : '<Root>/Sine Wave' */ 0.0 , /* SineWave_Phase : '<Root>/Sine Wave' */ 5.0 /* Gain_Gain : '<Root>/Gain' */ }; . . . non_tunable_sin_Y.Out1 = rtb_u * non_tunable_sin_P.Gain_Gain; </pre>
On	<pre> non_tunable_sin_Y.Out1 = rtb_u * 5.0; </pre>

Tunable Parameter Storage

A *tunable* parameter is a block parameter whose value can be changed at run-time. A tunable parameter is inherently noninlined. A *tunable expression* is an expression that contains one or more tunable parameters.

When you declare a parameter tunable, you control whether or not the parameter is stored within *model_P*. You also control the symbolic name of the parameter in the generated code.

When you declare a parameter tunable, you specify

- The *storage class* of the parameter.

In Real-Time Workshop, the storage class property of a parameter specifies how Real-Time Workshop declares the parameter in generated code.

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

- A *storage type qualifier*, such as `const` or `volatile`. This is simply a string that is included in the variable declaration, without error checking.
- (Implicitly) the symbolic name of the variable or field in which the parameter is stored. Real-Time Workshop derives variable and field names from the names of tunable parameters.

Real-Time Workshop generates a variable or struct storage declaration for each tunable parameter. Your choice of storage class controls whether the parameter is declared as a member of `model_P` or as a separate global variable.

You can use the generated storage declaration to make the variable visible to your code. You can also make variables declared in your code visible to the generated code. You are responsible for properly linking your code to generated code modules.

You can use tunable parameters or expressions in your root model and in masked or unmasked subsystems, subject to certain restrictions. (See “Tunable Expressions” on page 5-13.)

To declare tunable parameters, you must first enable the **Inline parameters** option. You then use the **Model Parameter Configuration** dialog to remove individual parameters from inlining and declare them to be tunable. This allows you to improve overall efficiency by inlining most parameters, while at the same time retaining the flexibility of run-time tuning for selected parameters. Another way you can achieve the same result is by using Simulink data objects; see “Simulink Data Objects and Code Generation” on page 5-33 for specific details.

The mechanics of declaring tunable parameters are discussed in “Using the Model Parameter Configuration Dialog” on page 5-8.

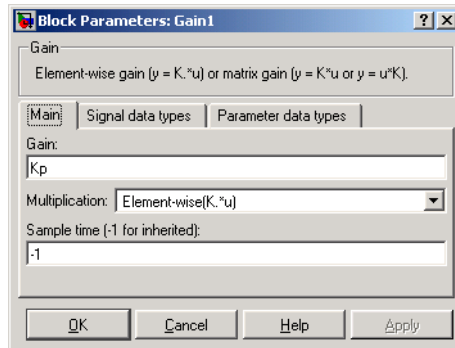
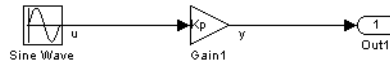
Storage Classes of Tunable Parameters

Real-Time Workshop defines four storage classes for tunable parameters. You must declare a tunable parameter to have one of the following storage classes:

- **SimulinkGlobal(Auto):** `SimulinkGlobal(Auto)` is the default storage class. Real-Time Workshop stores the parameter as a member of `model_P`. Each member of `model_P` is initialized to the value of the corresponding workspace variable at code generation time.
- **ExportedGlobal:** The generated code instantiates and initializes the parameter and `model.h` exports it as a global variable. An exported global variable is independent of the `model_P` data structure. Each exported global variable is initialized to the value of the corresponding workspace variable at code generation time.
- **ImportedExtern:** `model_private.h` declares the parameter as an extern variable. Your code must supply the proper variable definition and initializer, if any.
- **ImportedExternPointer:** `model_private.h` declares the variable as an extern pointer. Your code must supply the proper pointer variable definition and initializer, if any.

The generated code for `model.h` includes `model_private.h` in order to make the extern declarations available to subsystem files.

As an example of how the storage class declaration affects the code generated for a parameter, consider the model shown below.



The workspace variable `Kp` sets the gain of the `Gain1` block. Assume that the value of `Kp` is 3.14. The table “Tunable Parameter Storage Declarations and Code” on page 5-8 shows the variable declarations and the code generated for the gain block when `Kp` is declared as a tunable parameter. An example is shown for each storage class.

Note Real-Time Workshop uses column-major ordering for two-dimensional signal and parameter data. When interfacing your hand-written code interfaces to such signals or parameters via `ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer` declarations, make sure that your code observes this ordering convention.

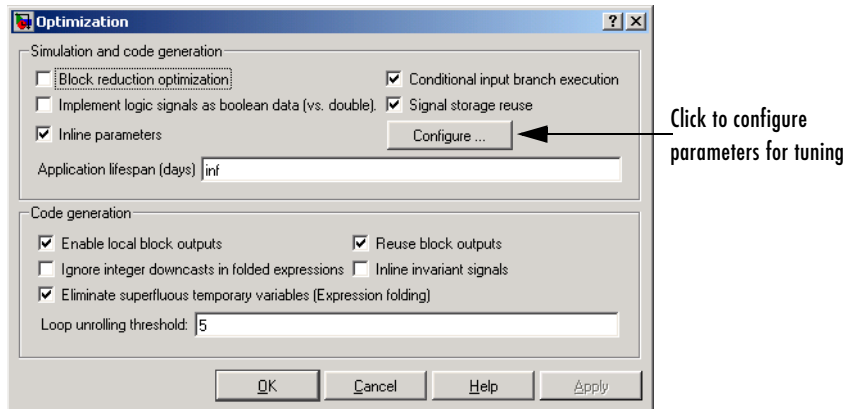
Note that the symbolic name Kp is preserved in the variable and field names in the generated code.

Tunable Parameter Storage Declarations and Code

Storage Class	Generated Variable Declaration and Code
SimulinkGlobal (Auto)	<pre> typedef struct _Parameters_tunable_sin Parameters_tunable_sin; struct _Parameters_tunable_sin { real_T Kp; }; Parameters_tunable_sin tunable_sin_P = { 3.14 }; . . tunable_sin_Y.Out1 = rtb_u * tunable_sin_P.Kp; </pre>
ExportedGlobal	<pre> real_T Kp = 3.14; . . tunable_sin_Y.Out1 = rtb_u * Kp; </pre>
ImportedExtern	<pre> extern real_T Kp; . . tunable_sin_Y.Out1 = rtb_u * Kp; </pre>
ImportedExtern Pointer	<pre> extern real_T *Kp; . . tunable_sin_Y.Out1 = rtb_u * (*Kp); </pre>

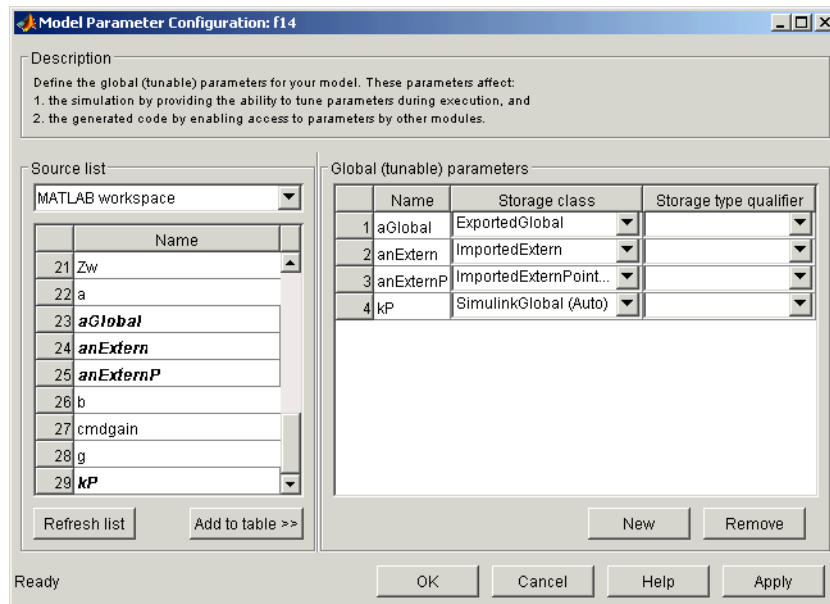
Using the Model Parameter Configuration Dialog

The **Model Parameter Configuration** dialog is available only when the **Inline parameters** option on the **Optimization** pane is selected. Selecting this option activates the **Configure** button, as shown below:



Clicking the **Configure** button opens the **Model Parameter Configuration** dialog.

Note The **Model Parameter Configuration** dialog has no capability to tune parameters within referenced models (models invoked by Model blocks). You can only tune parameters in referenced models by defining them in argument lists in the referenced models, and then declaring values for these parameter arguments in the Model block dialogs.



The Model Parameter Configuration Dialog

The **Model Parameter Configuration** dialog lets you select base workspace variables and declare them to be tunable parameters in the current model. The dialog is divided into two panels:

- The **Global (tunable) parameters** panel displays and maintains a list of tunable parameters associated with the model.
- The **Source list** panel displays a list of workspace variables and lets you add them to the tunable parameters list.

To declare tunable parameters, you select one or more variables from the source list, add them to the **Global (tunable) parameters** list, and set their storage class and other attributes.

Source List Panel. The **Source list** panel displays a menu and a scrolling table of numerical workspace variables.

The menu lets you choose the source of the variables to be displayed in the list. There are two choices: MATLAB workspace (lists all variables in the MATLAB

workspace that have numeric values), and Referenced workspace variables (lists only those variables referenced by the model). The source list displays names of variables defined in the MATLAB base workspace.

Selecting one or more variables from the source list enables the **Add to table** button. Clicking **Add to table** adds selected variables to the tunable parameters list in the **Global (tunable) parameters** panel. This action is all that is necessary to declare tunable parameters. However, if a block parameter which is not tunable is set to the name that appears on this list, an error results during simulation.

In the source list, the names of variables added to the tunable parameters list are displayed in italics (See the preceding figure).

The **Refresh list** button updates the table of variables to reflect the current state of the workspace. If you define or remove variables in the workspace while the **Model Parameter Configuration** dialog is open, click the **Refresh list** button when you return to the dialog. The new variables are added to the source list.

Global (Tunable) Parameters Panel. The **Global (tunable) parameters** panel displays a scrolling table of variables that have been declared tunable in the current model and lets you specify their attributes. The **Global (tunable) parameters** panel also lets you remove entries from the list or create new tunable parameters.

You select individual variables and change their attributes directly in the table. The attributes are

- **Storage class** of the parameter in the generated code. Select one of
 - SimulinkGlobal(Auto)
 - ExportedGlobal
 - ImportedExtern
 - ImportedExternPointer

See “Storage Classes of Tunable Parameters” on page 5-6 for definitions.

- **Storage type qualifier** of the variable in the generated code. For variables with any storage class *except* 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 additional qualifiers to the list. Note that the code generator does not check the storage type

qualifier for validity. The code generator includes the qualifier string in the generated code without syntax checking.

- **Name** of the parameter. This field is used only when creating a new tunable variable.

The **Remove** button deletes selected variables from the **Global (tunable) parameters** list.

The **New** button lets you create new tunable variables in the **Global (tunable) parameters** list. At a later time, you can add references to these variables in the model.

If the name you enter matches the name of an existing workspace variable in the **Source list**, that variable is declared tunable, and is displayed in italics in the **Source list**.

To define a new tunable variable, click the **New** button. This creates an empty entry in the table. Then, enter the name and attributes of the variable and click **Apply**.

Note If you edit the name of an existing variable in the list, you actually create a new tunable variable with the new name. The previous variable is removed from the list and loses its tunability (that is, it is inlined).

Declaring Tunable Variables

To declare an existing variable tunable

- 1 Open the **Model Parameter Configuration** dialog.
- 2 In the **Source list** panel, click the desired variable in the list to select it.
- 3 Click the **Add to table** button. The variable then appears in the table of tunable variables in the **Global (tunable) parameters** panel.
- 4 Click the variable in the **Global (tunable) parameters** panel.
- 5 Select the desired storage class from the **Storage class** menu.

- 6 Optionally, select (or enter) a storage type qualifier, such as `const` or `volatile` that you want the variable to have.
- 7 Click **Apply**, or click **OK** to apply changes and close the dialog.

Tunable Expressions

Real-Time Workshop supports the use of tunable variables in expressions. An expression that contains one or more tunable parameters is called a *tunable expression*.

Currently, there are certain limitations on the use of tunable variables in expressions. When an expression described below as not supported is encountered during code generation, a warning is issued and a nontunable expression is generated in the code. The limitations on tunable expressions are

- Complex expressions are not supported, except where the expression is simply the name of a complex variable.
- The use of certain operators or functions in expressions containing tunable operands is restricted. Restrictions are applied to four categories of operators or functions, classified in the following table:

Tunability Classification of Operators and Functions

Category	Operators or Functions
1	+ - .* ./ < > <= >= == ~= &
2	* /
3	abs, acos, asin, atan, atan2, boolean, ceil, cos, cosh, exp, floor, int8, int16, int32, log, log10, rem, sign, sin, sinh, sqrt, tan, tanh, uint8, uint16, uint32
4	: . ^ ^ [] {} . \ .\ ' .' ; ,

The rules applying to each category are as follows:

- Category 1 is unrestricted. These operators can be used in tunable expressions with any combination of scalar or vector operands.

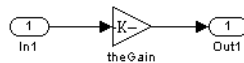
- Category 2 operators can be used in tunable expressions where at least one operand is a scalar. That is, scalar/scalar and scalar/matrix operand combinations are supported, but not matrix/matrix.
- Category 3 lists all functions that support tunable arguments. Tunable arguments passed to these functions retain their tunability. Tunable arguments passed to any other functions lose their tunability.
- Category 4 operators are not supported.

Note The “dot” (structure membership) operator is not supported. This means that expressions that include a structure member are not tunable.

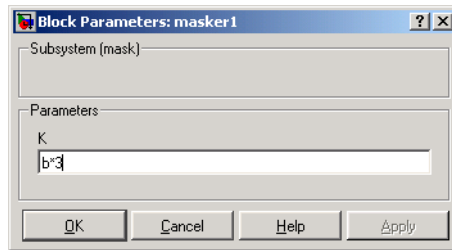
Tunable Expressions in Masked Subsystems

Tunable expressions are allowed in masked subsystems. You can use tunable parameter names or tunable expressions in a masked subsystem dialog. When referenced in lower-level subsystems, such parameters remain tunable.

As an example, consider the masked subsystem depicted below. The masked dialog variable `k` sets the gain parameter of `theGain`.



Suppose that the base workspace variable `b` is declared tunable with `SimulinkGlobal(Auto)` storage class. The following figure shows the tunable expression `b*3` in the subsystem’s mask dialog.



Tunable Expression in Subsystem Mask Dialog

Real-Time Workshop produces the following output computation for theGain. The variable *b* is represented as a member of the global parameters structure, *model_P*. (Note that for clarity in showing the individual Gain block computation, **Expression folding** was turned off in this example.)

```
/* Gain: '<S1>/theGain' */
rtb_theGain_C = rtb_SineWave_n * (( subsys_mask_P.b * 3.0 ));

/* Outport: '<Root>/Out1' */
subsys_mask_Y.Out1 = rtb_theGain_C;
```

As this example illustrates, for GRT targets, the parameter structure is mangled to create the structure identifier *model_P* (subject to the identifier length constraint). This is done to avoid namespace clashes in combining code from multiple models using model reference. ERT-based targets provide ways to customize identifier names.

When **Expression folding** is turned on, the above code condenses to

```
/* Outport: '<Root>/Out1' incorporates:
 * Gain: '<S1>/theGain'
 */
subsys_mask_Y.Out1 = rtb_SineWave_n * (( subsys_mask_P.b * 3.0 ));
```

Limitations of Tunable Expressions in Masked Subsystems. Expressions that include variables that were declared or modified in mask initialization code are *not* tunable.

As an example, consider the subsystem above, modified as follows:

- The mask initialization code is

```
t = 3 * k;
```

- The parameter k of the myGain block is $4 + t$.
- Workspace variable $b = 2$. The expression $b * 3$ is plugged into the mask dialog as in the preceding figure.

Since the mask initialization code can only run once, k is evaluated at code generation time as

```
4 + (3 * (2 * 3) )
```

Real-Time Workshop inlines the result. Therefore, despite the fact that b was declared tunable, the code generator produces the following output computation for theGain. (Note that for clarity in showing the individual Gain block computation, **Expression folding** was turned off in this example.)

```
/* Gain Block: <S1>/theGain */  
rtb_temp0 *= (22.0);
```

Other Limitations on Tunable Expressions

The Fcn block does not support tunable expressions in code generation.

Tunability of Linear Block Parameters

The following blocks have a Realization parameter that affects the tunability of their parameters:

- Transfer Fcn
- State-Space
- Discrete Transfer Fcn
- Discrete State-Space
- Discrete Filter

The Realization parameter must be set via the MATLAB `set_param` command, as in the following example.

```
set_param(gcf, 'Realization', 'auto')
```

The following values are defined for the Realization parameter:

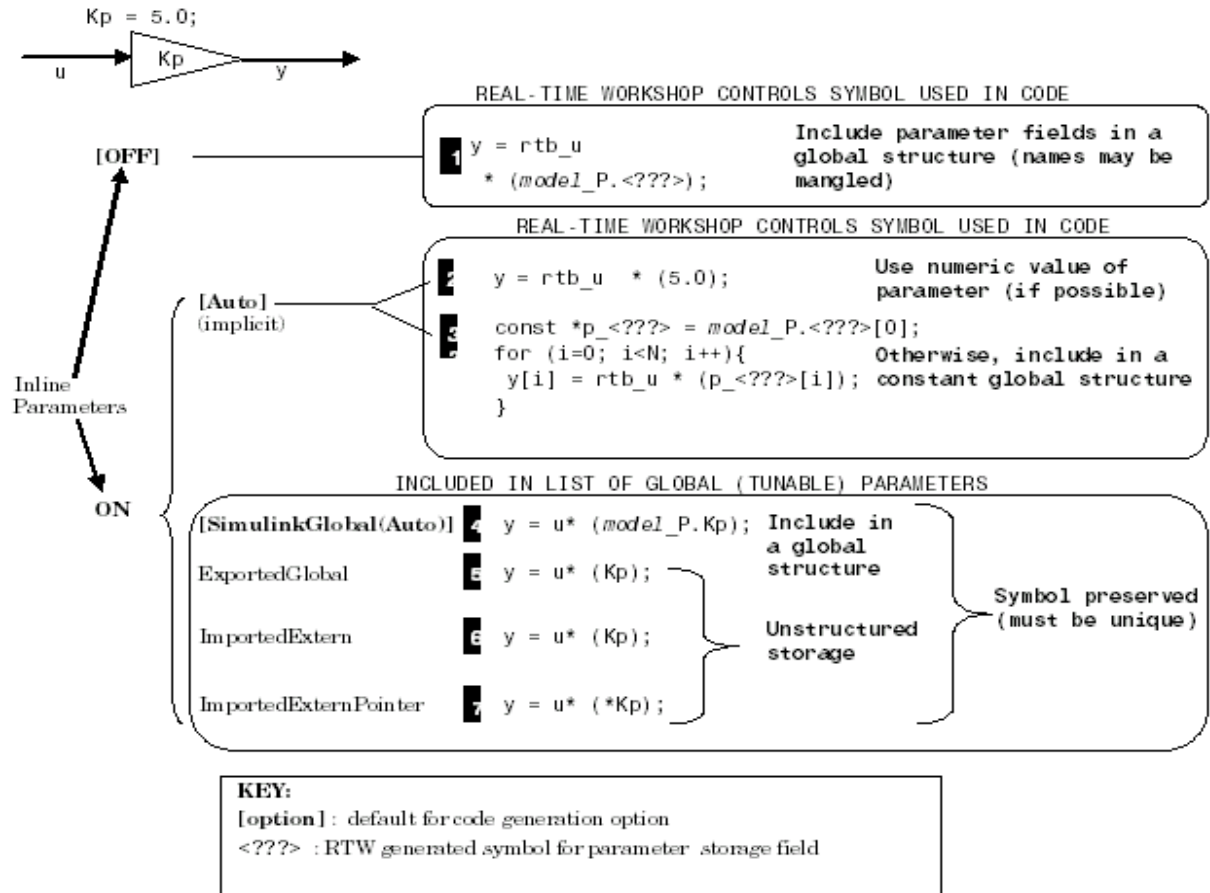
- **general**: The block's parameters are preserved in the generated code, permitting parameters to be tuned.

- **sparse:** The block's parameters are represented in the code by transformed values that increase the computational efficiency. Because of the transformation, the block's parameters are no longer tunable.
- **auto:** This setting is the default. A general realization is used if one or more of the block's parameters are tunable. Otherwise sparse is used.

Note To tune the parameter values of a block of one of the above types without restriction during an external mode simulation, you must set Realization to general.

Parameter Configuration Quick Reference Diagram

Figure illustrates the code generation and storage class options that control the representation of parameters in generated code.



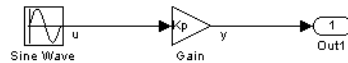
Parameter Configuration Quick Reference Diagram

Signal Storage, Optimization, and Interfacing

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

- Control whether signal storage is declared in global memory space or locally in functions (i.e., in stack variables).
- Control the allocation of stack space when using local storage.
- Ensure that particular signals are stored in unique memory locations by declaring them as *test points*.
- Reduce memory usage by instructing Real-Time Workshop 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 `Signals_examp`, the model shown in the figure below.



Signals_examp Model

Signal Storage Concepts

This section discusses structures and concepts you must understand in order 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 in Real-Time Workshop

The Global Block I/O Structure

By default, Real-Time Workshop attempts to optimize memory usage by sharing signal memory and using local variables.

However, there are a number of circumstances in which it is desirable or necessary to 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 (or all) 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 fragment illustrates how *model_B* is defined and declared in code generated (with signal storage optimizations off) from the *Signals_examp* model shown in Figure .

```
(in model.h)
/* Block signals (auto storage) */
typedef struct _BlockIO_signals_examp {
    real_T SinSig;                /* '<Root>/Sine Wave' */
} BlockIO_signals_examp;
.
.
.
(in model.c)
/* Block signals (auto storage) */
BlockIO_signals_examp signals_examp_B;
.
.
.
(in model.c; gain is optimized to be a local variable)
/* local block i/o variables */

real_T rtb_Gain1Sig;
```

Field names for signals stored in *model_B* are generated according to the rules described in “Symbolic Naming Conventions for Signals in Generated Code” on page 5-29.

Storage Classes for Signals

In Real-Time Workshop, the *storage class* property of a signal specifies how Real-Time Workshop declares and stores the signal. In some cases this specification is qualified by further options.

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

Default Storage Class. Auto is the default storage class. Auto is the appropriate storage class 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 block outputs**, and **Enable local block outputs** options, and on available stack space. See “Signals with Auto Storage Class” on page 5-22 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 *model_B*, or in unstructured global variables, independent of *model_B*. These storage classes are appropriate for signals that you want to monitor and/or interface to external code.

The **Signal storage reuse** and **Local block outputs** optimizations do not apply to signals with storage classes other than Auto.

Use the **Signal Properties** dialog to assign these storage classes to signals:

- **SimulinkGlobal(Test Point):** Test points are stored as fields of the *model_B* structure that are not shared or reused by any other signal. See “Declaring Test Points” on page 5-25 for further information.
- **ExportedGlobal:** The signal is stored in a global variable, independent of the *model_B* data structure. *model_private.h* exports the variable. Signals with **ExportedGlobal** storage class must have unique signal names. See “Interfacing Signals to External Code” on page 5-27 for further information.
- **ImportedExtern:** *model_private.h* declares the signal as an extern variable. Your code must supply the proper variable definition. Signals with

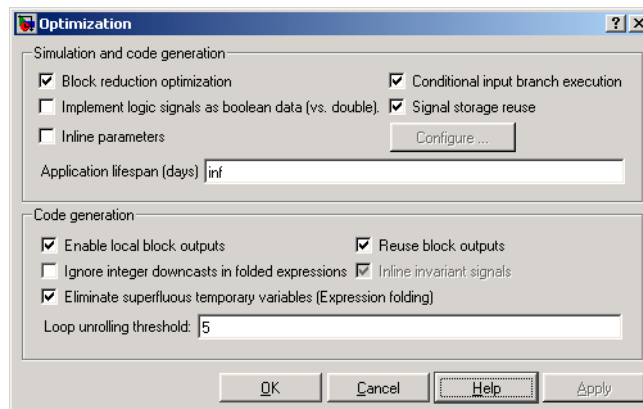
ImportedExtern storage class must have unique signal names. See “Interfacing Signals to External Code” on page 5-27 for further information.

- ImportedExternPointer: *model_private.h* declares the signal as an extern pointer. Your code must supply a proper pointer variable definition. Signals with ImportedExtern storage class must have unique signal names. See “Interfacing Signals to External Code” on page 5-27 for further information.

Signals with Auto Storage Class

This section discusses options that are available for signals with Auto storage class. These options let you control signal memory reuse and choose local or global (*model_B*) storage for signals.

The **Signal storage reuse** and **Reuse block outputs** options control signal memory reuse. The **Signal storage reuse** option is on the **Optimization** pane of the **Configuration Parameters** dialog box, as shown below:



When **Signal storage reuse** is on, the **Enable local block outputs** and **Reuse block outputs** options in the Code Generation section of the dialog become visible, and are selected by default.

Note that these options interact. When the **Signal storage reuse** option is on,

- The **Reuse block outputs** option is enabled. By default, **Reuse block outputs** is on and signal memory is reused whenever possible.

- The **Enable Local block outputs** option is enabled. This lets you choose whether reusable signal variables are declared as local variables in functions or as members of *model_B*.

The following code examples illustrate the effects of the **Signal storage reuse**, **Reuse block outputs**, and **Enable local block outputs** options. The examples were generated from the `Signals_examp` model (see figure “Signals_examp Model” on page 5-19).

The first example illustrates maximal signal storage optimization, with **Signal storage reuse**, **Reuse block outputs**, and **Enable local block outputs** on (the default). The output signals from the Sine Wave and Gain blocks reuse `rtb_SinSig`, a variable local to the `Md1Outputs` function.

```
/* local block i/o variables */

real_T rtb_SinSig;
real_T rtb_Gain1Sig;

/* Sin Block: '<Root>/Sine Wave' */

rtb_SinSig = signals_examp_P.SineWave_Amp *
    sin(signals_examp_P.SineWave_Freq *
        signals_examp_M->Timing.t[0] +
        signals_examp_P.SineWave_Phase) +
    signals_examp_P.SineWave_Bias;

/* Gain: '<Root>/Gain' */
rtb_Gain1Sig = rtb_SinSig * signals_examp_P.Gain_Gain;

/* Output: '<Root>/Out1' */
signals_examp_Y.Out1 = rtb_Gain1Sig;
```

If you are constrained by limited stack space, you can turn **Enable local block outputs** off and still benefit from memory reuse. The following example was generated with **Enable local block outputs** off and **Signal storage reuse** and **Reuse block outputs** on. The output signals from the Sine Wave and Gain blocks use global structure `signals_examp_B` rather than declaring local variables.

```
signals_examp_B.SinSig = signals_examp_P.SineWave_Amp *
    sin(signals_examp_P.SineWave_Freq *
```

```
signals_examp_M->Timing.t[0] +
signals_examp_P.SineWave_Phase) +
signals_examp_P.SineWave_Bias;

/* Gain: '<Root>/Gain' */
signals_examp_B.Gain1Sig = signals_examp_B.SinSig *
signals_examp_P.Gain_Gain;

/* Outport: '<Root>/Out1' */
signals_examp_Y.Out1 = signals_examp_B.Gain1Sig;
```

When the **Signal storage reuse** option is off, **Reuse block outputs** and **Enable local block outputs** are disabled. This makes all block outputs global and unique, as in the following code fragment.

```
/* Sin Block: <Root>/Sine Wave */

signals_examp_B.SinSig = signals_examp_P.SineWave_Amp *
sin(signals_examp_P.SineWave_Freq *
signals_examp_M->Timing.t[0] +
signals_examp_P.SineWave_Phase) +
signals_examp_P.SineWave_Bias;

/* Gain: '<Root>/Gain' */
signals_examp_B.Gain1Sig = signals_examp_B.SinSig *
signals_examp_P.Gain_Gain;

/* Outport: '<Root>/Out1' */
signals_examp_Y.Out1 = signals_examp_B.Gain1Sig;
```

In large models, disabling **Signal storage reuse** can significantly increase RAM and ROM usage. Therefore, this approach is not recommended.

The following table summarizes the possible combinations of the **Signal storage reuse/ Reuse block outputs** and **Enable local block outputs** options.

Global, Local, and Reusable Signal Storage Options

	Signal storage reuse and Reuse block outputs ON	Signal storage reuse OFF (Reuse 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 <i>model_B</i> structure	Individual signal storage in <i>model_B</i> structure

Controlling Stack Space Allocation

When the **Local block outputs** option is on, the use of stack space is constrained by the following TLC variables:

- **MaxStackSize**: The total allocation size of local variables that are declared by all functions in the entire model cannot exceed **MaxStackSize** (in bytes). **MaxStackSize** can be any positive integer. If the total size of local variables exceeds this maximum, the Target Language Compiler allocates the remaining variables in global, rather than local, memory.
The default value for **MaxStackSize** is `rtInf`, i.e. unlimited stack size.
- **MaxStackVariableSize**: Limits the size of any local variable declared in a function to *N* bytes, where *N*>0. A variable whose size exceeds **MaxStackVariableSize** is allocated in global, rather than local, memory.

You can change the values of these variables in your system target file if necessary. See “Target Language Compiler Variables and Options” on page 2-77 for further information.

Declaring Test Points

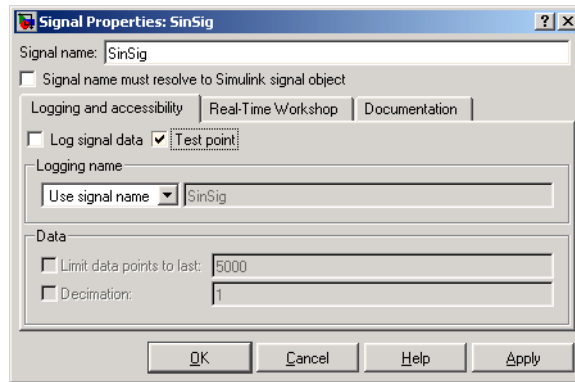
A *test point* is a signal that is stored in a unique location that is not shared or reused by any other signal. *Test-pointing* is the process of declaring a signal to be a test point.

You declare a signal to be a test point as follows:

- 1 Select the signal line by right-clicking it.
- 2 From the context menu that pops up, select **Signal properties**.

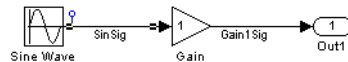
Alternatively, after selecting the line that carries the signal, choose **Signal Properties** from the **Edit** menu of your model. This also opens the **Signal Properties** dialog box.

- 3 On the **Logging and accessibility** tab, select the **Test point** option on the **Signal properties** dialog, as shown below:



- 4 Click **OK** to dismiss the dialog.

A stemmed-circle icon appears on the signal line to indicate the test point, as shown below:



Test points are stored as members of the *model_B* structure, even when the **Signal storage reuse** and **Local block outputs** option are selected. Test-pointing lets you override these options for individual signals. Therefore,

you can test-point selected signals without losing the benefits of optimized storage for the other signals in your model.

For an example of storage declarations and code generated for a test point, see “Summary of Signal Storage Class Options” on page 5-30.

Interfacing Signals to External Code

The Simulink **Signal Properties** dialog lets you interface selected signals to externally written code. In this way, your hand-written code has access to such signals for monitoring or other purposes. To interface a signal to external code, use the **Real-Time Workshop** tab of the **Signal Properties** dialog box to assign one of the following storage classes to the signal:

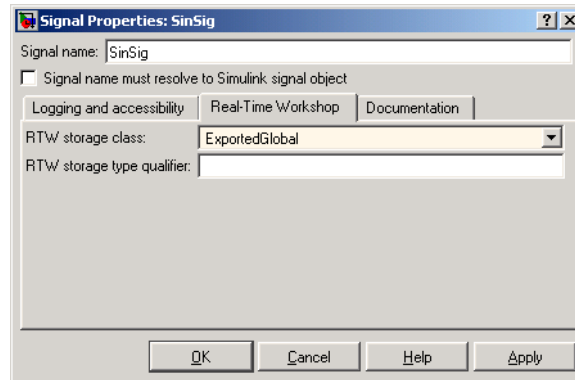
- ExportedGlobal
- ImportedExtern
- ImportedExternPointer

Set the storage class as follows:

- 1** In your Simulink block diagram, select the line that carries the signal. Then select **Signal Properties** from the **Edit** menu of your model. This opens the **Signal Properties** dialog box.

Alternatively, right-click the line that carries the signal, and select **Signal properties** from the pop-up menu.

- 2** Select the **Real-Time Workshop** tab of the **Signal properties** dialog.
- 3** Select the desired storage class (Auto, ExportedGlobal, ImportedExtern, or ImportedExternPointer) from the **RTW storage class** menu. The figure below shows ExportedGlobal selected:



4 Optional: For storage classes other than Auto, you can enter a storage type qualifier such as `const` or `volatile` in the **RTW storage type qualifier** field. Note that Real-Time Workshop does not check this string for errors; whatever you enter is included in the variable declaration.

5 Click **Apply**.

Note You can also interface test points and other signals that are stored as members of `model_B` to your code. To do this, your code must know the address of the `model_B` structure where the data is stored, and other information. This information is not automatically exported. Real-Time Workshop provides C and Target Language Compiler APIs that give your code access to `model_B` and other data structures. See “C-API for Interfacing with Signals and Parameters” on page 17-2 for further information.

Limitation on Stateflow Outputs

Note that a nonscalar output signal exiting a Stateflow chart cannot be assigned storage class `ImportedExternPointer`.

Symbolic Naming Conventions for Signals in Generated Code

When signals have a storage class other than Auto, Real-Time Workshop 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.SinSig* and *model_B.Gain1Sig* are derived from the corresponding labeled signals in the *Signals_exam* model (shown in figure “Signals_exam Model” on page 5-19).

```
/* Block signals (auto storage) */
typedef struct _BlockIO_signals_exam {
    real_T SinSig;                /* '<Root>/Sine Wave' */
    real_T Gain1Sig;             /* '<Root>/Gain' */
} BlockIO_signals_exam;
```

Note that when the optimization **Signal Storage Reuse** is off, *Gain1Sig* 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 naming format is

```
model_B.system#_BlockName_outport#
```

where *system#* is a unique system number assigned by Simulink, *BlockName* is the name of the source block, and *outport#* is a port number. The port number (*outport#*) is used only when the source block or subsystem has multiple output ports.

When a signal has Auto storage class, Real-Time Workshop controls generation of variable or field names without regard to signal labels.

Summary of Signal Storage Class Options

The table below shows, for each signal storage class option, the variable declaration and the code generated for Sine Wave output (SinSig) of the model shown in figure “Signals_examp Model” on page 5-19.

Signal Properties Options and Generated Code

Storage Class	Declaration	Code
Auto (with storage optimizations on)	<code>real_T rtb_SinSig;</code> (in <i>model.c</i>)	<code>rtb_SinSig = signals_examp_P.SineWave_Amp * sin(signals_examp_P.SineWave_Freq * signals_examp_M->Timing.t[0] + signals_examp_P.SineWave_Phase) + signals_examp_P.SineWave_Bias;</code>
Test point	<code>typedef struct _BlockIO_model { real_T SinSig; real_T Gain1Sig; } BlockIO; (in <i>model.h</i>)</code> <code>. BlockIO_model model_B; (in <i>model.c</i>)</code>	<code>signals_examp_B.SinSig = signals_examp_P.SineWave_Amp * sin(signals_examp_P.SineWave_Freq * signals_examp_M->Timing.t[0] + signals_examp_P.SineWave_Phase) + signals_examp_P.SineWave_Bias;</code>
Exported Global	<code>extern real_T SinSig;</code> (in <i>model.h</i>) <code>real_T SinSig;</code> (in <i>model.c</i>)	<code>SinSig = signals_examp_P.SineWave_Amp * sin(signals_examp_P.SineWave_Freq * signals_examp_M->Timing.t[0] + signals_examp_P.SineWave_Phase) + signals_examp_P.SineWave_Bias;</code>

Signal Properties Options and Generated Code

Storage Class	Declaration	Code
Imported Extern	extern real_T SinSig; (in <i>model_private.h</i>)	SinSig = signals_examp_P.SineWave_Amp * sin(signals_examp_P.SineWave_Freq * signals_examp_M->Timing.t[0] + signals_examp_P.SineWave_Phase) + signals_examp_P.SineWave_Bias;
Imported Extern Pointer	extern real_T *SinSig; (in <i>model_private.h</i>)	(*SinSig) = signals_examp_P.SineWave_Amp * sin(signals_examp_P.SineWave_Freq * signals_examp_M->Timing.t[0] + signals_examp_P.SineWave_Phase) + signals_examp_P.SineWave_Bias;

C API for Parameter Tuning and Signal Monitoring

Real-Time Workshop includes support for development of a C application program interface (API) for tuning parameters and monitoring signals independent of external mode. See “C-API for Interfacing with Signals and Parameters” on page 17-2 for information.

Target Language Compiler API for Parameter Tuning and Signal Monitoring

Real-Time Workshop includes support for development of a Target Language Compiler API for tuning parameters and monitoring signals independent of external mode. See “Target Language Compiler API for Signals and Parameters” on page 17-16 for information.

Parameter Tuning via MATLAB Commands

The **Model Parameter Configuration** dialog is the recommended way to see or set the attributes of tunable parameters. However, you can also use MATLAB `get_param` and `set_param` commands.

The following commands return the tunable parameters and/or their attributes:

- `get_param(gcs, 'TunableVars')`

- `get_param(gcs, 'TunableVarsStorageClass')`
- `get_param(gcs, 'TunableVarsTypeQualifier')`

The following commands declare tunable parameters or set their attributes:

- `set_param(gcs, 'TunableVars', str)`
The argument `str` (string) is a comma-separated list of variable names.
- `set_param(gcs, 'TunableVarsStorageClass', str)`
The argument `str` (string) 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` (string) 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`.

```
set_param(gcs, 'TunableVars', 'k1')
set_param(gcs, 'TunableVarsStorageClass', 'ExportedGlobal')
set_param(gcs, 'TunableVarsTypeQualifier', 'const')
```


Simulink Data Objects and Code Generation

Before using Simulink data objects with Real-Time Workshop, read the following:

- The discussion of Simulink data objects in Using Simulink
- “Parameters: Storage, Interfacing, and Tuning” on page 5-2
- “Signal Storage, Optimization, and Interfacing” on page 5-19

Within the class hierarchy of Simulink data objects, Simulink provides two classes that are designed as base classes for signal and parameter storage:

- `Simulink.Parameter`: Objects that are instances of the `Simulink.Parameter` class or any class derived from `Simulink.Parameter` are called *parameter objects*.
- `Simulink.Signal`: Objects that are instances of the `Simulink.Signal` class or any class derived from `Simulink.Signal` are called *signal objects*.

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

Real-Time Workshop also writes information about the properties of parameter and signal objects to the `model.rtw` file. This information, formatted as `Object` records, is accessible to Target Language Compiler programs. For general information on `Object` records, see “Object information in the `model.rtw` file” in the Target Language Compiler Reference Guide.

The general procedure for using Simulink data objects in code generation is as follows:

- 1** Define a subclass of one of the built-in `Simulink.Data` classes.
 - For parameters, define a subclass of `Simulink.Parameter`.
 - For signals, define a subclass of `Simulink.Signal`.
- 2** Instantiate parameter or signal objects from your subclass and set their properties appropriately, using the Simulink Data Explorer.
- 3** Use the objects as parameters or signals within your model.

4 Generate code and build your target executable.

The following sections describe the relationship between Simulink data objects and code generation in Real-Time Workshop.

Parameter Objects

This section discusses how to use parameter objects in code generation.

Configuring Parameter Objects for Code Generation

In configuring parameter objects for code generation, you use the following code generation and parameter object properties:

- The **Inline parameters** option (see “Parameters: Storage, Interfacing, and Tuning” on page 5-2).
- Parameter object properties:
 - **Value.** This property is the numeric value of the object, used as an initial (or inlined) parameter value in generated code.
 - **RTWInfo.StorageClass.** This property controls the generated storage declaration and code for the parameter object.

Other parameter object properties (such as user-defined properties of classes derived from `Simulink.Parameter`) do not affect code generation.

Note If **Inline parameters** is off (the default), the `RTWInfo.StorageClass` parameter object property is ignored in code generation.

Effect of Storage Classes on Code Generation for Parameter Objects

Real-Time Workshop generates code and storage declarations based on the `RTWInfo.StorageClass` property of the parameter object. The logic is as follows:

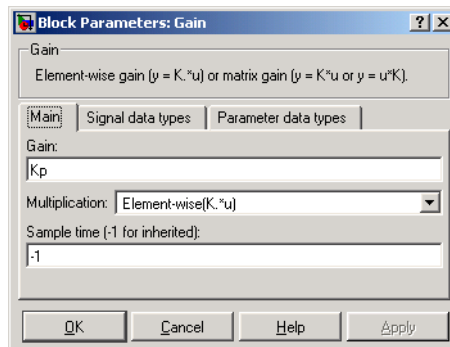
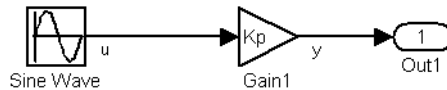
- If the storage class is 'Auto' (the default), the parameter object is inlined (if possible), using the `Value` property.
- For storage classes other than 'Auto', the parameter object is handled as a tunable parameter.

- A global storage declaration is generated. You can use the generated storage declaration to make the variable visible to your hand-written code. You can also make variables declared in your hand-written code visible to the generated code.
- The symbolic name of the parameter object is preserved in the generated code.

See Table for examples of code generated for each possible setting of `RTWInfo.StorageClass`.

Example of Parameter Object Code Generation

In this section, the Gain block computations of the model shown in the figure below are used as an example of how Real-Time Workshop generates code for a parameter object.



Model Using Parameter Object Kp As Block Parameter

In this model, Kp sets the gain of the Gain1 block.

To configure a parameter object such as Kp for code generation,

- 1 Instantiate a `Simulink.Parameter` object called `Kp`. In this example, the parameter object is an instance of the example class `SimulinkDemos.Parameter`, which is provided with `Simulink`.

```
Kp = Simulink.Parameter
Kp =
Simulink.Parameter
    RTWInfo: [1x1 Simulink.ParamRTWInfo]
  Description: ''
    DocUnits: ''
        Min: -Inf
        Max: Inf
    Value: 5
    DataType: 'auto'
  Complexity: 'real'
  Dimensions: '[1x1]'
```

Make sure that the name of the parameter object matches the desired block parameter in your model. This ensures that `Simulink` can associate the parameter name with the correct object. In the preceding model, the Gain block parameter `Kp` resolves to the parameter object `Kp`.

- 2 Set the object properties you need. You can do this via the Model Explorer, or you can assign properties via MATLAB commands, as follows:

- To specify the `Value` property, type

```
Kp.Value = 5.0;
```

- To specify the storage class of for the parameter, set the `RTWInfo.StorageClass` property, for example:

```
Kp.RTWInfo.StorageClass = 'ExportedGlobal';
```

The `RTWInfo` parameters are now

```
Kp.RTWInfo
Simulink.ParamRTWInfo
    StorageClass: 'ExportedGlobal'
        Alias: ''
  CustomStorageClass: 'Default'
  CustomAttributes: [1x1
SimulinkCSC.AttribClass_Simulink_Default]
```

The following table shows the variable declarations for Kp and the code generated for the Gain block in the model shown in the preceding model, with **Inline parameters** on and **expression folding** on (which includes the gain computation in the output computation). An example is shown for each possible setting of `RTWInfo.StorageClass`. Note that global structures include the model name (symbolized as *model_* or *_model*).

Code Generation from Parameter Objects (Inline Parameters ON)

StorageClass Property	Generated Variable Declaration and Code
Auto	<pre>model_Y.Out1 = rtb_u * 5.0;</pre>
Simulink Global	<pre>struct _Parameters_model { real_T Kp; } . . Parameters_model model_P = { 5.0 }; . . model_Y.Out1 = rtb_u * model_P.Kp;</pre>
Exported Global	<pre>extern real_T Kp; . . real_T Kp = 5.0; . . model_Y.Out1 = rtb_u * Kp;</pre>

Code Generation from Parameter Objects (Inline Parameters ON)

StorageClass Property	Generated Variable Declaration and Code
Imported Extern	<pre>extern real_T Kp; . . model_Y.Out1 = rtb_u * Kp;</pre>
Imported Extern Pointer	<pre>extern real_T *Kp; . . model_Y.Out1 = rtb_u * (*Kp);</pre>

Signal Objects

This section discusses how to use signal objects in code generation.

Configuring Signal Objects for Code Generation

In configuring signal objects for code generation, you use the following code generation options and signal object properties:

- The **Signal storage reuse** code generation option (see “Signal Storage, Optimization, and Interfacing” on page 5-19).
- The **Enable local block outputs** code generation option (see “Signal Storage, Optimization, and Interfacing” on page 5-19).
- The `RTWInfo.StorageClass` signal object property: The storage classes defined for signal objects, and their effect on code generation, are the same for model signals and signal objects (see “Storage Classes for Signals” on page 5-21).

Other signal object properties (such as user-defined properties of classes derived from `Simulink.Signal`) do not affect code generation.

Effect of Storage Classes on Code Generation for Signal Objects

The way in which Real-Time Workshop uses storage classes to determine how signals are stored is the same with and without signal objects. However, if a

signal's label resolves to a signal object, the object's `RTWInfo.StorageClass` property is used in place of the port configuration of the signal.

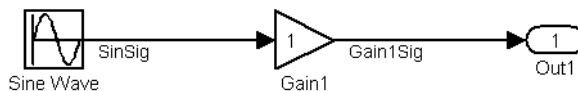
The default storage class is `Auto`. If the storage type is `Auto`, Real-Time Workshop follows the **Signal storage reuse**, **Buffer reuse**, and **Local block outputs** code generation options to determine whether signal objects are stored in reusable and/or local variables. Make sure that these options are set correctly for your application.

To generate a test point or externally interfaceable signal storage declaration, use an explicit `RTWInfo.StorageClass` assignment. For example, setting the storage class to `SimulinkGlobal`, as in the following command, is equivalent to declaring a signal as a test point.

```
SinSig.RTWInfo.StorageClass = 'SimulinkGlobal';
```

Example of Signal Object Code Generation

The discussion and code examples in this section refers to the model shown in the following figure:



Example Model With Signal Object

To configure a signal object, you must first create it and associate it with a labeled signal in your model. To do this,

- 1 Define a subclass of `Simulink.Signal`. In this example, the signal object is an instance of the example class `SimulinkDemos.Signal`, which is provided with Simulink. For the definition of `SimulinkDemos.Signal`, see the directory `matlabroot/toolbox/simulink/simdemos/@SimulinkDemos`.
- 2 Instantiate a signal object from your subclass. The following example instantiates `SinSig`, a signal object of class `SimulinkDemos.Signal`.

```
SinSig = SimulinkDemos.Signal;
```

Make sure that the name of the signal object matches the label of the desired signal in your model. This ensures that Simulink can resolve the signal label to the correct object. For example, in the model shown in the above figure, the signal label SinSig would resolve to the signal object SinSig.

- 3 Set the object properties as required. You can do this via the Simulink Data Explorer. Alternatively, you can assign properties via MATLAB commands. For example, assign the signal object's storage class by setting the RTWInfo.StorageClass property as follows.

```
SinSig.RTWInfo.StorageClass = 'ExportedGlobal';
```

The following table shows, for each setting of RTWInfo.StorageClass, the variable declaration and the code generated for Sine Wave output (SinSig) of the model shown in the above figure.

Signal Properties Options and Generated Code

Storage Class	Declaration	Code
Auto (with storage optimizations on)	<pre>real_T rtb_SinSig; (in model.c)</pre>	<pre>rtb_SinSig = signals_examp_P.SineWave_Amp * sin(signals_examp_P.SineWave_Freq * signals_examp_M->Timing.t[0] + signals_examp_P.SineWave_Phase) + signals_examp_P.SineWave_Bias;</pre>
Simulink Global	<pre>typedef struct _BlockIO_model { real_T SinSig; } BlockIO_model; (in model.h) . . BlockIO_model model_B; (in model.c)</pre>	<pre>signals_examp_B.SinSig = signals_examp_P.SineWave_Amp * sin(signals_examp_P.SineWave_Freq * signals_examp_M->Timing.t[0] + signals_examp_P.SineWave_Phase) + signals_examp_P.SineWave_Bias;</pre>

Signal Properties Options and Generated Code

Storage Class	Declaration	Code
Exported Global	<pre>real_T SinSig; (in <i>model.c</i>) extern real_T SinSig; (in <i>model.h</i>)</pre>	<pre>SinSig = signals_examp_P.SineWave_Amp * sin(signals_examp_P.SineWave_Freq * signals_examp_M->Timing.t[0] + signals_examp_P.SineWave_Phase) + signals_examp_P.SineWave_Bias;</pre>
Imported Extern	<pre>extern real_T SinSig; (in <i>model_private.h</i>)</pre>	<pre>SinSig = signals_examp_P.SineWave_Amp * sin(signals_examp_P.SineWave_Freq * signals_examp_M->Timing.t[0] + signals_examp_P.SineWave_Phase) + signals_examp_P.SineWave_Bias;</pre>
Imported Extern Pointer	<pre>extern real_T *SinSig; (in <i>model_private.h</i>)</pre>	<pre>(*SinSig) = signals_examp_P.SineWave_Amp * sin(signals_examp_P.SineWave_Freq * signals_examp_M->Timing.t[0] + signals_examp_P.SineWave_Phase) + signals_examp_P.SineWave_Bias;</pre>

Resolving Conflicts in Configuration of Parameter and Signal Objects

This section describes how to avoid and resolve certain conflicts that can arise when using parameter and signal objects.

Parameters

The next two figures illustrate a case where both a tunable parameter K_p (declared in the **Model Parameter Configuration** dialog box) and an identically named parameter object K_p (defined from the command line or in **Model Explorer**) exist. If K_p is used as a block parameter, there is a potential for ambiguity when Simulink attempts to resolve the symbol K_p .

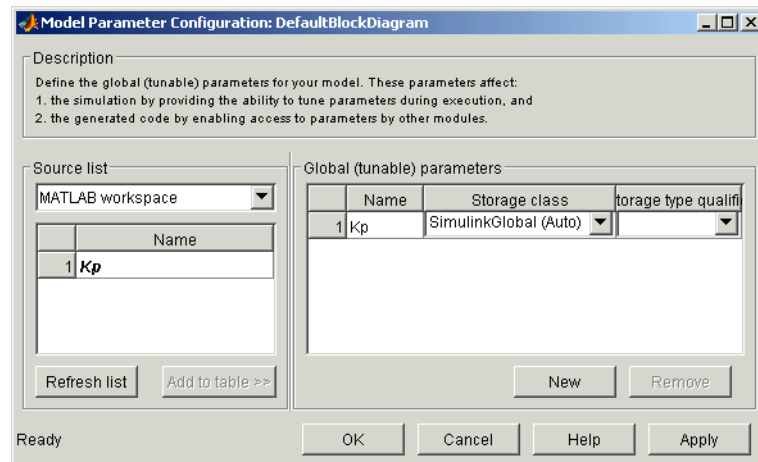


Figure 5-1: Parameter K_p Defined with SimulinkGlobal Storage Class

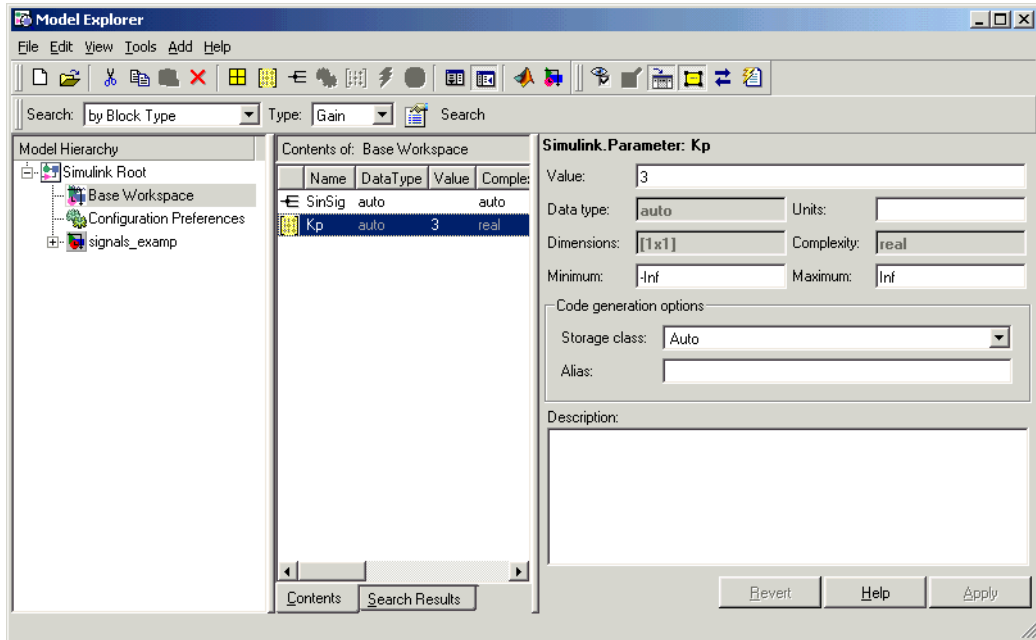


Figure 5-2: Parameter Object Kp with Auto Storage Class in Model Explorer

An obvious solution would be to assign different names to the parameter and the parameter object.

If this is not desirable, however, you should make sure that the storage class properties of identically named parameters and parameter objects are compatible in accordance with the following table. If they are not, an error message is displayed when the model is run, and/or when code generation is initiated.

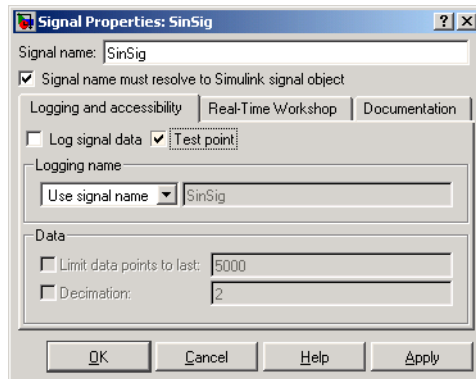
In the previous two figures, the parameter Kp has SimulinkGlobal(auto) storage class and the parameter object Kp has Auto storage class. Accordingly, the symbol Kp would resolve to the parameter object Kp.

Compatible Parameter/Parameter Object Storage Class Configurations

Tunable Parameter		Parameter Object	
	Auto (default)	SimulinkGlobal	Other
Auto (default)	Use parameter object	Use parameter object	Use parameter object
SimulinkGlobal	Error	Use parameter object	Error
Other	Error	Error	If storage class and TypeQualifier same, use parameter object, otherwise error

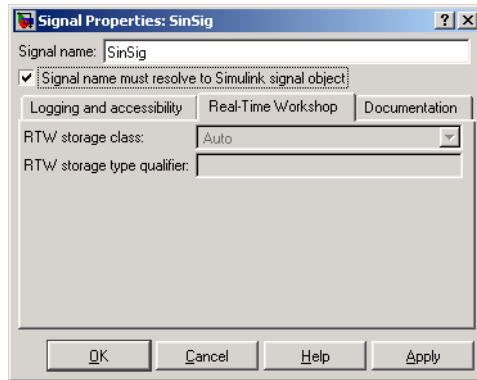
Signals and Block States

The following illustrates a case where both a signal Sig (defined in the **Signal Properties** dialog box) and a signal object Sig (defined via the command line or in **Model Explorer**) exist. There is a potential for ambiguity when Simulink attempts to resolve the symbol Sig.



Signal SinSig Defined as SimulinkGlobal (Test Point)

As the preceding figure shows, one way to resolve the ambiguity is to specify that a signal must resolve to a Simulink data object. You do this by selecting the top check box in the **Signal properties** dialog. When you do this, you no longer can specify the **RTW storage class** property in the **Signal properties** dialog, as the figure below illustrates:



The RTW storage class menu is disabled, because it is up to the SinSig Simulink.Signal object to specify its own storage class.

An obvious solution would be to assign different names to the signal and the signal object. If this is not desirable, however, you should make sure that the storage class properties of identically named signals and signal objects are compatible in accordance with Table . If they are not, an error message will be displayed when model is run, and/or when code generation is initiated.

The signal and signal objects Sig both have SimulinkGlobal storage class. Therefore no conflict would arise, and Sig would resolve to the signal object Sig.

Note The rules for compatibility between block states/signal objects are identical to those given for signals/signal objects.

Signal	Signal Object		
	Auto (default)	SimulinkGlobal	Other
Auto (default)	Use signal object	Use signal object	Use signal object
SimulinkGlobal (Test Point)	Error	Use signal object	Error
Other	Error	Error	If storage class and TypeQualifier are the same, use signal object, otherwise error

Compatible Signal/Signal Object Configurations

Customizing Code for Parameter and Signal Objects

You can further influence the treatment of parameter and signal objects in generated code by using TLC to access fields in object records in *model.rtw* files. For details on doing this, see “Object information in the *model.rtw* file” in the Target Language Compiler Reference Guide.

Using Objects to Export ASAP2 Files

The ASAM-ASAP2 Data Definition Target provides special signal and parameter subclasses that support exporting of signal and parameter object information to ASAP2 data files. For information about the ASAP2 target and its associated classes and TLC files, see Appendix B, “Generating ASAP2 Files.”

Block States: Storing and Interfacing

For certain block types, Real-Time Workshop lets you control how block states in your model are stored and represented in the generated code. Using the **State properties** tab of a block dialog, you can

- Control whether or not states declared in generated code are interfaceable (visible) to externally written code. You can also specify that states are to be stored in locations declared by externally written code.
- Assign symbolic names to block states in generated code.

Storage of Block States

The discussion of block state storage in this section applies to the following block types:

- Discrete Filter
- Discrete State-Space
- Discrete-Time Integrator
- Discrete Transfer Function
- Discrete Zero-Pole
- Memory
- Unit Delay

These block types 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. For further information on the DWork vector, see “Block State and Work Vector Functions” in the Target Language Compiler Reference Guide.

If you want to interface a block state to your hand-written code, you can specify that the state is to be stored in a location other than the DWork vector. You do this by assigning a storage class to the block state.

You can also define a symbolic name, to be used in code generation, for a block state.

Block State Storage Classes

The storage class property of a block state specifies how Real-Time Workshop declares and stores the state in a variable. Storage class options for block states are similar to those for signals. The available storage classes are

- Auto
- ExportedGlobal
- ImportedExtern
- ImportedExternPointer

Default Storage Class

Auto is the default storage class. Auto is the appropriate storage class for states that you do not need to interface to external code. States with Auto storage class are stored as members of the Dwork vector.

You can assign a symbolic name to states with Auto storage class. If you do not supply a name, Real-Time Workshop generates one, as described in “Symbolic Names for Block States” on page 5-50.

Explicitly Assigned Storage Classes

Block states with storage classes other than Auto are stored in unstructured global variables, independent of the Dwork vector. These storage classes are appropriate for states that you want to interface to external code. The following storage classes are available for states:

- ExportedGlobal: The state is stored in a global variable. *model_private.h* exports the variable. States with ExportedGlobal storage class must have unique names.
- ImportedExtern: *model_private.h* declares the state as an extern variable. Your code must supply the proper variable definition. States with ImportedExtern storage class must have unique names.
- ImportedExternPointer: *model_private.h* declares the state as an extern pointer. Your code must supply the proper pointer variable definition. States with ImportedExternPointer storage class must have unique names.

The table “State Properties Options and Generated Code” on page 5-55, gives examples of variable declarations and the code generated for block states with each type of storage class.

You can assign a symbolic name to states with any of the above storage classes. If you do not supply a name, Real-Time Workshop generates one, as described in “Symbolic Names for Block States” on page 5-50.

The next section describes how to use the **State Properties** dialog to assign storage classes to block states.

Using the State Properties Dialog to Interface States to External Code

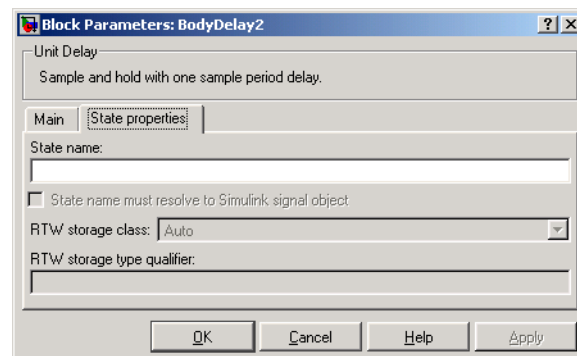
The **State Properties** tab of the relevant blocks’ parameter dialogs lets you interface a block’s state to external code by assigning a storage class other than Auto (i.e., ExportedGlobal, ImportedExtern, or ImportedExternPointer) to the state.

Set the storage class as follows:

- 1 In your Simulink block diagram, select the desired block. Then select **<blockname> properties** from the **Edit** menu. This opens the block dialog box, containing two or more tabs, one of which is **State properties**.

Alternatively, you can right-click the block and select **Block properties** from the pop-up context menu.

- 2 Click the **State Properties** tab. The default **State Properties** settings are shown below:



- 3 Enter a name for the variable to be used to store block state in the **State name** field.

- 4 Click **Apply** to register the variable name.

The first two fields beneath the **State name**, **State name must resolve to Simulink signal object** and **RTW storage class**, become enabled.

- 5 If the state is to be stored in a Simulink signal object in the base or model workspace, select **State name must resolve to Simulink signal object**.
If you choose this option, you cannot declare a storage class for the state, and the field below becomes disabled.
- 6 Select the desired storage class (ExportedGlobal, ImportedExtern, or ImportedExternPointer) from the **RTW storage class** menu.
- 7 *Optional:* For storage classes other than Auto, you can enter a storage type qualifier such as `const` or `volatile` in the **RTW storage type qualifier** field. Note that Real-Time Workshop does not check this string for errors; whatever you enter is included in the variable declaration.
- 8 Click **OK** or **Apply** and close the dialog box.

Symbolic Names for Block States

To determine the variable or field name generated for a block's state, you can either

- Use a default name generated by Real-Time Workshop
- Define a symbolic name via the **State Name** field of the **State Properties** dialog

Default Block State Naming Convention

If you do not define a symbolic name for a block state, Real-Time Workshop uses the following default naming convention:

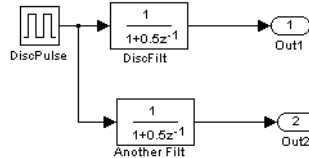
```
BlockType#_DSTATE
```

where

- BlockType is the name of the block type (e.g., `Discrete_Filter`).
- # is a unique ID number (#) assigned by Real-Time Workshop if multiple instances of the same block type appear in the model. The ID number is appended to BlockType.

- `_DSTATE` is a string that is always appended to the block type and ID number.

For example, consider the model shown in the following figure:



Model with Two Discrete Filter Block States

Examine code generated for the states of the two Discrete Filter blocks. Assume that:

- Neither block's 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 fragment below:

```

/* 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;
    }
}

```

User-Defined Block State Names

Using the **State Properties** dialog, you can define your own symbolic name for a block state. To do this,

- 1 Select the desired block. Then select **<blockname> properties** from the **Edit** menu. This opens the block dialog box, containing two or more tabs, one of which is **State properties**.

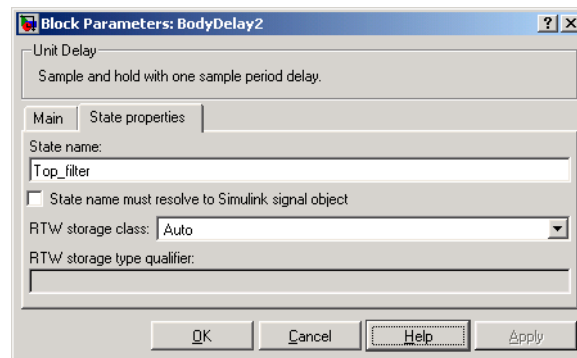
Alternatively, you can right-click the block and select **Block properties** from the pop-up context menu.

- 2 Click the **State properties** tab.

- 3 Enter the symbolic name in the **State name** field of the **State Properties** dialog. For example, enter the state name `Top_filter`.

The **State name** field turns yellow to indicate that you changed it.

- 4 Click **Apply**. The dialog appears as follows:



- 5 Click **OK** or **Cancel** to dismiss the block dialog.

The following state initialization code was generated from the example model shown in figure “Example Model With Signal Object” on page 5-39, 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`.

```
disc_filt_statesR14_DWork.Top_filter = 0.0;

/* 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;
```

Block States and Simulink Signal Objects

If you are not familiar with Simulink data objects and signal objects, you should read “Simulink Data Objects and Code Generation” on page 5-33 before reading this section.

You can associate a block state with a signal object, and control code generation for the block state through the signal object. To do this,

- 1 Instantiate the desired signal object, and set its `RTWInfo.StorageClass` property as you require.
- 2 Open the **State Properties** dialog box for the block whose state you want to associate with the signal object. Enter the name of the signal object in the **State name** field.
- 3 Make sure that the storage class and type qualifier settings of the block’s **State Properties** dialog box are compatible with those of the signal object. See “Resolving Conflicts in Configuration of Parameter and Signal Objects” on page 5-42.
- 4 Click **Apply** and close the dialog box.

Note When a block state is associated with a signal object, the mapping between the block state and the signal object must be one to one. If two or more identically named entities, such as a block state and a signal, map to the same signal object, the name conflict is flagged as an error at code generation time.

Summary of State Storage Class Options

The following table shows, for each state storage class option, the variable declaration and MdlInitialize code generated for a Discrete Filter block state. The block state has the user-defined state name `Lower_filter`.

State Properties Options and Generated Code

Storage Class	Declaration	Initialization Code
Auto	<pre>typedef struct D_Work_DiscFilt_tag { real_T Lower_filter; } D_Work_DiscFilt; (in model.h) . . D_Work_DiscFilt DiscFilt_DWork; (in model.c)</pre>	<pre>DiscFilt_DWork.Lower_filter = DiscFilt_U.In1 + (DiscFilt_P.DiscreteFilter_A)*Disc Filt_DWork.Lower_filter;</pre>
Exported Global	<pre>real_T Lower_filter; (in model.c) extern real_T Lower_filter; (in model.h)</pre>	<pre>Lower_filter = DiscFilt_U.In1 + (DiscFilt_P.DiscreteFilter_A) * Lower_filter; (in model.c)</pre>
Imported Extern	<pre>extern real_T Lower_filter; (in model_private.h)</pre>	<pre>Lower_filter = DiscFilt_U.In1 + (DiscFilt_P.DiscreteFilter_A) * Lower_filter; (in model.c)</pre>
Imported Extern Pointer	<pre>extern real_T *Lower_filter; (in model_private.h)</pre>	<pre>(*Lower_filter) = DiscFilt_U.In1 + (DiscFilt_P.DiscreteFilter_A) * (*Lower_filter); (in model.c)</pre>

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 block name.

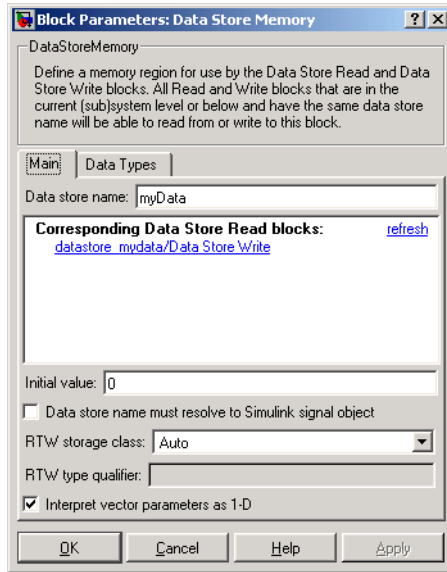
Note that you can generate code from multiple Data Store Memory blocks that have the same name, subject to the following restriction: *at most one* of the identically-named blocks can have a storage class other than Auto. An error is reported if this condition is not met. For blocks with Auto storage class, Real-Time Workshop generates a unique symbolic name for each block (if necessary) to avoid name clashes. For blocks with storage classes other than Auto, Real-Time Workshop simply uses the block name to generate the symbol.

To control the storage declaration for a Data Store Memory block, use the **RTW storage class** and **RTW storage type qualifier** fields of the Data Store Memory block parameters dialog.

In the following block diagram, a Data Store Write block writes to memory declared by the Data Store Memory block myData.



Data Store Memory blocks are nonvirtual, because code is generated for their initialization and declarations in model header files. The Data Store Memory block parameter dialog is shown next. Note that it documents the blocks that write to and read from it.



The following table shows code generated for the Data Store Memory block in this model, depending on the setting of **RTW storage class**. The table gives the variable declarations and MdlOutputs code generated for the myData block.

Storage Class Options for Data Store Memory Blocks and Generated Code

Storage Class	Declaration	Code
Auto	<pre>typedef struct D_Work_tag { real_T myData; } D_Work; (declared in <i>model.h</i>) . . /* Block states (auto storage) */ D_Work <i>model_DWork</i>; (declared in <i>model.c</i>)</pre>	<pre><i>model_DWork</i>.myData = rtb_SineWave;</pre>
Exported Global	<pre>/* Exported block states */ real_T myData; (declared in <i>model.c</i>) extern real_T myData; (declared in <i>model.h</i>)</pre>	<pre>myData = rtb_SineWave;</pre>
Imported Extern	<pre>extern real_T myData; (declared in <i>model_private.h</i>)</pre>	<pre>myData = rtb_SineWave;</pre>
Imported Extern Pointer	<pre>extern real_T *myData; (declared in <i>model_private.h</i>)</pre>	<pre>(*myData) = rtb_SineWave;</pre>

Data Store Memory and Simulink Signal Objects

If you are not familiar with Simulink data objects and signal objects, you should read “Simulink Data Objects and Code Generation” on page 5-33 before reading this section.

You can associate a Data Store Memory block with a signal object, and control code generation for the block through the signal object. To do this,

- 1 Instantiate the desired signal object, and set its `RTWInfo.StorageClass` property as you require.

- 2** Open the block parameters dialog box for the Data Store Memory block whose state you want to associate with the signal object. Enter the name of the signal object in the **Data store name** field.
- 3** Make sure that the storage class and type qualifier settings of the block parameters dialog box are compatible with those of the signal object. See “Resolving Conflicts in Configuration of Parameter and Signal Objects” on page 5-42.
- 4** Click **Apply** and close the dialog box.

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.

External Mode

In external mode, Real-Time Workshop establishes a communications link between a model running in Simulink and code executing on a target system. Further details on external mode are provided elsewhere in this documentation: Chapter 17, “Creating an External Mode Communication Channel,” contains advanced information for those who want to implement their own external mode communications layer. You want to read it to gain increased insight into the architecture and code structure of external mode communications. In addition, Chapter 13, “Targeting Tornado for Real-Time Applications,” discusses the use of external mode in the VxWorks Tornado environment.

Introduction (p. 6-2)	Summary of external mode features and architecture
Using the External Mode User Interface (p. 6-3)	Describes all elements of the external mode user interface
External Mode Compatible Blocks and Subsystems (p. 6-22)	Types of blocks that receive and view signals in external mode
External Mode Communications Overview (p. 6-25)	Summary of the communications process between Simulink and the target program
Client/Server Implementations (p. 6-28)	Features, bundled targets, and techniques for using external mode protocol via TCP/IP
Limitations of External Mode (p. 6-35)	External mode restrictions imposed by the structure of a model

Introduction

External mode allows two separate systems — a *host* and a *target* — to communicate. The host is the computer where MATLAB and Simulink are executing. The target is the computer where the executable created by Real-Time Workshop runs.

The host (Simulink) 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 Simulink is the client and the target is the server.

External mode lets you

- Modify, or *tune*, block parameters in real time. In external mode, whenever you change parameters in the block diagram, Simulink automatically downloads them to the executing target program. This lets you tune your program's parameters without recompiling. In external mode, the Simulink model becomes a graphical front end to the target program.
- View and log block outputs in many types of blocks and subsystems. You can monitor and/or 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 could be triggered by a selected signal crossing zero in a positive direction. Alternatively, you can manually trigger data uploading.

External mode works by establishing a communication channel between Simulink and code generated by Real-Time Workshop. This channel is implemented by a low-level *transport layer* that handles physical transmission of messages. Both Simulink and the generated model code are independent of this layer. The transport layer and the 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 ERT, GRT, GRT malloc, and RSim targets support external mode host/target communication via TCP/IP and RS-232 (serial) communication. xPC targets use a customized transport layer, which is neither TCP/IP nor serial. The Tornado target supports TCP/IP only. Note that serial transport is implemented only for Windows 32-bit architectures. The Real-Time Windows Target implements external mode communication via shared memory.

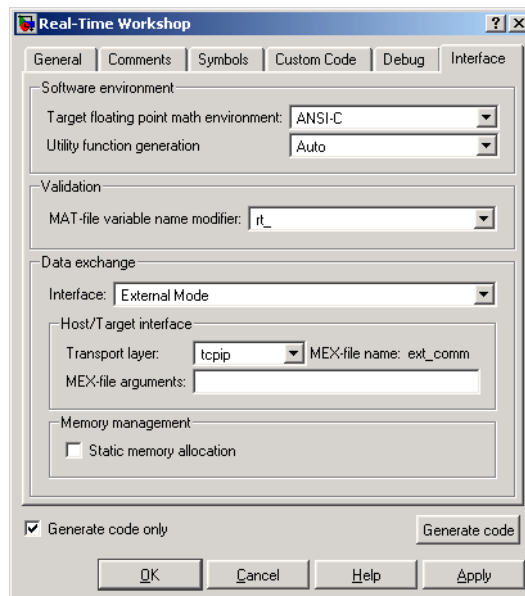
Using the External Mode User Interface

This section discusses the elements of the Simulink and Real-Time Workshop user interface that control operation of external mode. These elements include

- External mode-related menu items in **Simulation** and **Tools** menus and in the Simulink toolbar, including **Configuration Parameters** target options
- **External Mode Control Panel**
- **Target Interfacing**
- **External Signal Uploading and Triggering**
- **Data Archiving**

External Mode Interface Options

The ERT, GRT, GRT malloc, RSim, Real-Time Windows, and Tornado targets support external mode. All targets that support it feature a set of external mode options on their respective target tab of the **Configuration Parameters** dialog (this tab is normally named **Interface**). The following illustration is from the GRT target dialog, and is discussed below.



The **Data exchange** section at the bottom has the following elements:

- **Interface** menu: Selects which of three mutually exclusive data interfaces to include in the generated code. Options are
 - None
 - C-API
 - External mode
 - ASAP2

This chapter discusses only the External mode option. For information on other options, see “Interface Options” on page 2-42.

Once you select External mode from the Interface menu, the following options appear beneath:

- **Transport layer** pop-up menu: Identifies messaging protocol for host/target communications; choices are `tcpip` and `serial_win32`.

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: Type a list of arguments to be passed to the transport layer MEX-file; these will vary according to the protocol you use.

For more information on the transport options, see “Target Interfacing” on page 6-12 and “Client/Server Implementations” on page 6-28. You can add other transport protocols yourself by following instructions given in “Creating an External Mode Communication Channel” on page 17-18.

- **Static memory allocation** check box: Controls how memory for external mode communication buffers in the target is allocated. When you select this option, the following one appears beneath it:
- **Static memory buffer size** text field: Number of bytes to preallocate for external mode communications buffers in the target when **Static memory allocation** is used.

Note Selecting External mode from the **Interface** menu does not cause Simulink to operate in external mode (see “External Mode Related Menu and Toolbar Items” on page 6-5, below). Its function is to instrument the code generated for the target to support external mode.

The **Static memory allocation** check box (for GRT and ERT targets) directs Real-Time Workshop to generate code for external mode that uses only static memory allocation (“malloc-free” code). When selected, it activates the **Static memory buffer size** edit field, in which you specify the size of the static memory buffer used by external mode. The default value is 1,000,000 bytes. Should 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.

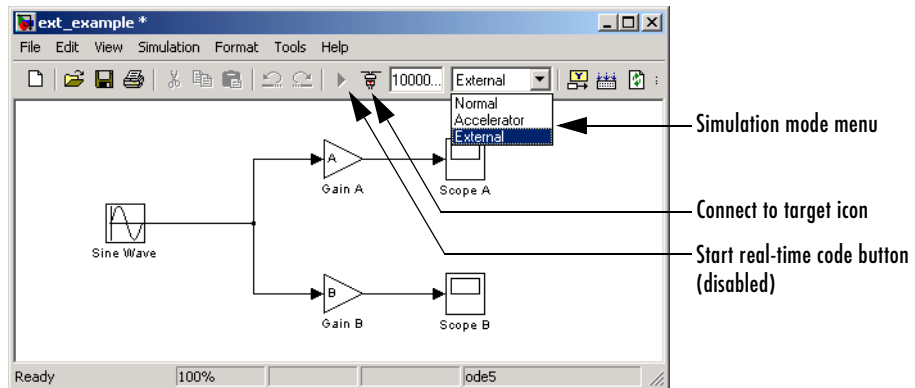
Hint To determine how much memory you need to allocate, enable verbose mode on the target (by including `OPTS=" -DVERBOSE "` on the make command line). As it executes, external mode displays the amount of memory it tries to allocate and the amount of memory available to it each time it attempts an allocation. Should an allocation fail, this console log can be used to adjust the size entered in the **Static memory buffer size** field.

Note When you create an ERT target, external mode can generate pure integer code. Select this feature with the **Integer code only** check box on the **ERT Target** tab of the **Configuration parameters** dialog. The option ensures that all code, including external mode support code, is free of doubles and floats. For further details, see the Real-Time Workshop Embedded Coder documentation.

External Mode Related Menu and Toolbar Items

To communicate with a target program, the model must be operating in external mode. The **Simulation** menu and the toolbar provide two ways to enable external mode:

- Select **External** from the **Simulation** menu.
- Select **External** from the simulation mode menu in the toolbar. The simulation mode menu is shown in this picture.



Simulation Mode Menu Options and Target Connection Control (Host Disconnected from Target)

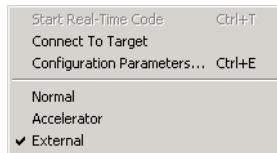
Once external mode is enabled, you can use the **Simulation** menu or the toolbar to connect to and control the target program.

Selecting external mode in the model window controls execution only, and does *not* cause Real-Time Workshop to generate code for external mode. To do this, you must select **External** mode from the **Interface** menu on the **Interface** tab of the **Configuration Parameters** dialog, as described above.

Note You can enable external mode, and simultaneously connect to the target system, by using the **External Mode Control Panel**. See “External Mode Control Panel” on page 6-10.

Simulation Menu

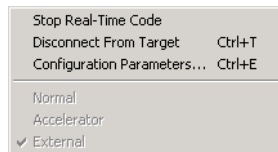
When Simulink is in external mode, the upper section of the **Simulation** menu contains external mode options. Initially, Simulink is disconnected from the target program, and the menu displays the options shown in this picture.



Simulation Menu External Mode Options (Host Disconnected from Target)

The **Connect to target** option establishes communication with the target program. When a connection is established, the target program might be executing model code, or it might be awaiting a command from the host to start executing model code. You can also accomplish this by clicking the Connect to target icon, as shown in the preceding figure.

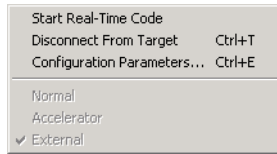
If the target program is executing model code, the **Simulation** menu contents change, as shown in this picture:



Simulation Menu External Mode Options (Target Executing Model Code)

The **Disconnect from target** option disconnects Simulink from the target program, which continues to run. The **Stop real-time code** option terminates execution of the target program and disconnects Simulink from the target system.

If the target program is in a wait state, the **Start real-time code** option is enabled, as shown in this picture. The **Start real-time code** option instructs the target program to begin executing the model code.



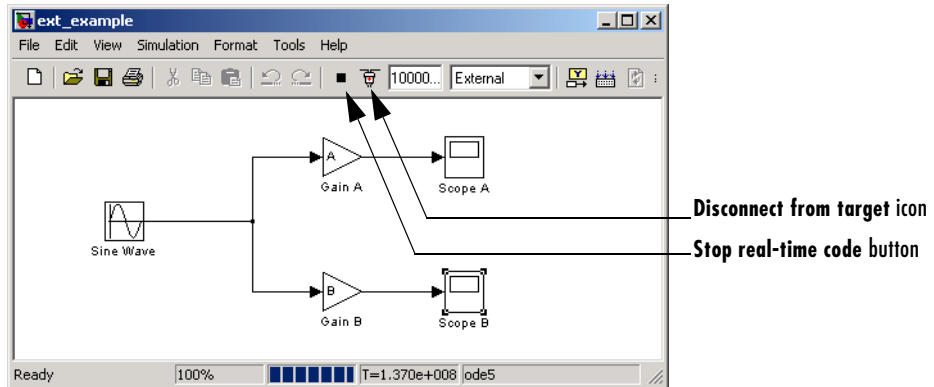
Simulation Menu External Mode Options (Target Awaiting Start Command)

Toolbar Controls

The Simulink toolbar controls, shown in Figure , Simulation Mode Menu Options and Target Connection Control (Host Disconnected from Target), let you control the same external mode functions as the **Simulation** menu. Simulink displays external mode icons to the left of the Simulation mode menu. Initially, the toolbar displays a Connect to target icon and a disabled **Start real-time code** button. Click the Connect to target icon to connect Simulink to the target program.

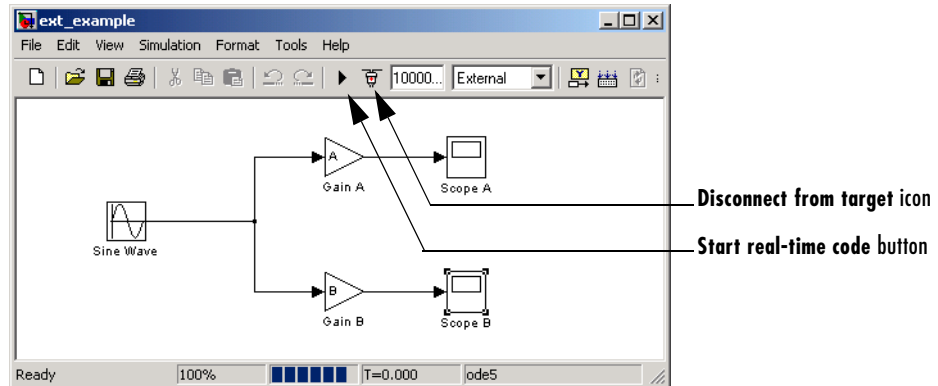
When a connection is established, the target program might be executing model code, or it might be awaiting a command from the host to start executing model code.

If the target program is executing model code, the toolbar displays a **Stop real-time code** button and a Disconnect from target icon (shown in Figure , External Mode Control Panel in Batch Download Mode). Click the **Stop real-time code** button to command the target program to stop executing model code and disconnect Simulink from the target system. Click the Disconnect from target icon to disconnect Simulink from the target program while leaving the target program running.



External Mode Toolbar Controls (Target Executing Model Code)

If the target program is in a wait state, the toolbar displays a **Start real-time code** button and a **Disconnect from target** icon (shown in Figure , External Mode Toolbar Controls (Target in Wait State)). Click the **Start real-time code** button to instruct the target program to start executing model code. Click the **Disconnect from target** icon to disconnect Simulink from the target program.



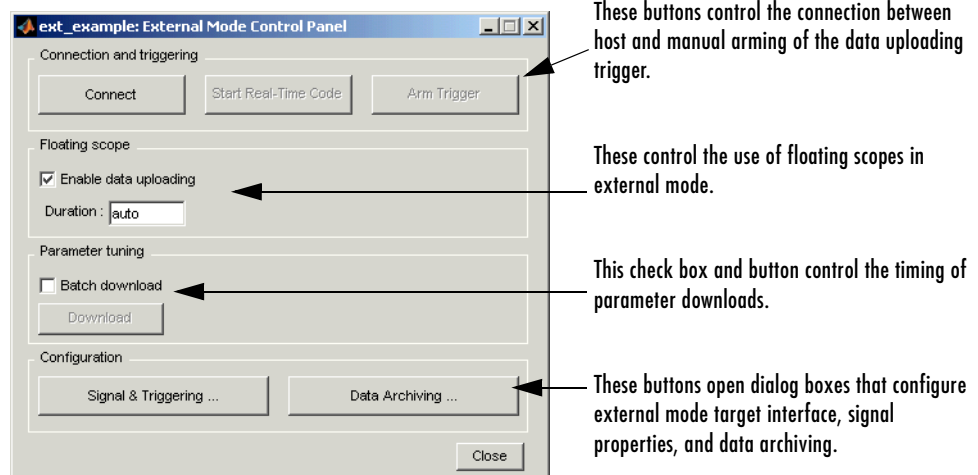
External Mode Toolbar Controls (Target in Wait State)

External Mode Control Panel

The **External Mode Control Panel**, illustrated below, provides centralized control of all external mode features, including

- Host/target connection, disconnection, and target program start/stop functions, and enabling of external mode
- Arming and disarming the data upload trigger
- External mode communications configuration
- Uploading data to Floating Scopes
- Timing of parameter downloads
- Selection of signals from the target program to be viewed and monitored on the host
- Configuration of data archiving features

Select **External mode control panel** from the Simulink **Tools** menu to open the **External Mode Control Panel**.



The following sections describe the features supported by the **External Mode Control Panel**.

Connecting, Starting, and Stopping

The **External Mode Control Panel** performs the same connect/disconnect and start/stop functions found in the **Simulation** menu and the Simulink toolbar (see “External Mode Related Menu and Toolbar Items” on page 6-5).

The **Connect/Disconnect** button connects to or disconnects from the target program. The button text changes in accordance with the state of the connection.

Note that if external mode is not enabled at the time the **Connect** button is clicked, the **External Mode Control Panel** enables external mode automatically.

The **Start/Stop real-time code** button commands the target to start or terminate model code execution. The button is disabled until a connection to the target is established. The button text changes in accordance with the state of the target program.

Floating Scope Options

The **Floating scope** pane of the **External Mode Control Panel** controls when and for how long data is uploaded to Floating Scope blocks. When used under external mode, Floating Scopes

- Do not appear in the signal and triggering GUI
- Support manual triggering only

Note that the behavior of wired scopes is not restricted in these ways.

The **Floating scope** pane contains a check box and an edit field:

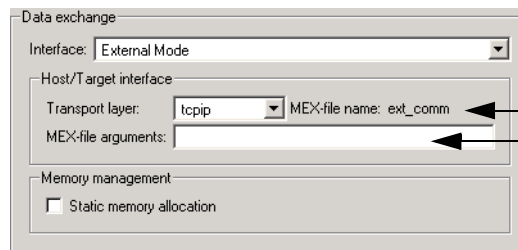
- **Enable data uploading** check box, which functions as an **Arm trigger** button for floating scopes. When the target is disconnected it controls whether or not to “arm when connect” the floating scopes. When already connected it acts as a toggle button to arm/cancel the trigger.
- **Duration** edit field, which specifies the duration for floating scopes. By default it is set to auto, which causes whatever value is specified in the signal and triggering GUI (which by default is 1000 seconds) to be used.

Target Interfacing

Real-Time Workshop lets you implement client and server transport for external mode using either TCP/IP or serial protocols. You can use the socket-based external mode implementation provided by Real-Time Workshop with the generated code, provided that your target system supports TCP/IP. Otherwise, use or customize the serial transport layer option provided.

A low-level *transport layer* handles physical transmission of messages. Both Simulink 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 **Host/Target interface** section of the **Interface** pane of the **Configuration Parameters** dialog. This section is shown below:



The external interface MEX-file being used (not editable; specified in system target file)

Type optional arguments to the external interface MEX-file here. This file must be in the current directory or in a directory that is on your MATLAB path.

The **Host/Target interface** controls let you verify the name of a MEX-file that implements host/target communications, and specify calling argument values. This is known as the external interface MEX-file. The meaning of these arguments depends on the MEX-file implementation.

The name of the external interface MEX-file is not editable. The default is `ext_comm`, the TCP/IP-based external interface file provided for use with the GRT, GRT malloc, ERT, RSIM, and Tornado targets. If you select the `serial_win32` transport option, the MEX-file name `ext_serial_win32_com` is displayed in this location. Custom or third-party targets can use a different external interface MEX-file. The interface MEX-file that appears in this field must be specified in the system target file.

- The **MEX-file arguments** edit field lets you specify parameters to the external interface MEX-file for communicating with executing targets.

For TCP/IP interfaces, `ext_comm` allows three optional arguments:

- The network name of your target
- Verbosity level (0 or 1)
- A TCP/IP server port number

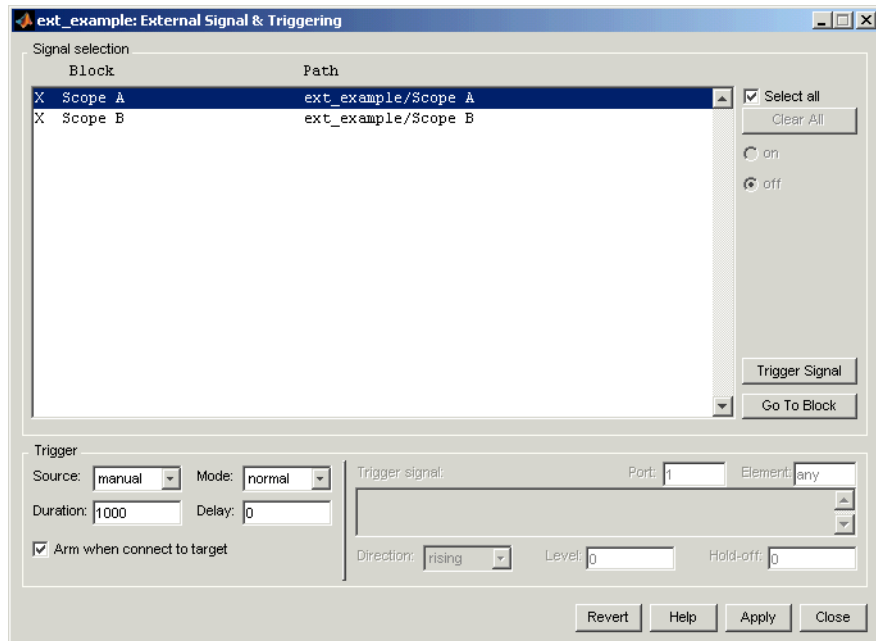
For serial transport, the optional arguments to `ext_serial_win32_comm` are

- Verbosity level (0 or 1)
- Serial port ID (e.g., 1 for COM1, etc.)
- Baud rate (selected from the set 1200, 2400, 4800, 9600, 14400, 19200, 38400, 57600, 115200, with a default baud rate of 57600)

See “Client/Server Implementations” on page 6-28 for details on MEX-file transport architecture and arguments.

External Signal Uploading and Triggering

Clicking the **Signal & triggering** button of the **External Mode Control Panel** activates the **External Signal & Triggering** dialog box, as shown below:



Default Settings of the External Signal & Triggering Dialog Box

The **External Signal & Triggering** dialog box displays a list of all blocks and subsystems in your model that support external mode signal uploading. See “External Mode Compatible Blocks and Subsystems” on page 6-22 for information on which types of blocks are external mode compatible.

The **External Signal & Triggering** dialog box lets you select the signals that are collected from the target system and viewed in external mode. It also lets you select a signal that triggers uploading of data when certain signal conditions are met, and define the triggering conditions.

Default Operation

Figure , Default Settings of the External Signal & Triggering Dialog Box shows the default settings of the **External Signal & Triggering** dialog box. The default operation of the **External Signal & Triggering** dialog box is designed to simplify monitoring the target program. If you use the default settings, you do not need to preconfigure signals and triggers. Simply start the target

program and connect the Simulink model to it. All external mode compatible blocks will be selected and the trigger will be armed. Signal uploading begins immediately upon connection to the target program.

The default configuration is

- **Arm when connect to target:** on
- **Trigger Mode:** normal
- **Trigger Source:** manual
- **Select all:** on

Signal Selection

All 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 to be viewed. An X appears to the left of each selected block's name.

The **Select all** check box selects all signals. By default, **Select all** is on.

If **Select all** is off, you can select or deselect individual signals using the **on** and **off** radio buttons. To select a signal, click the desired list entry and click the **on** radio button. To deselect a signal, click the desired list entry and click the **off** radio button. Alternatively, you can double-click a signal in the list to toggle between selection and deselection.

The **Clear all** button deselects all signals.

Trigger Options

The **Trigger** panel located at the bottom left of the **External Signal & Triggering** dialog box contains options that control when and how signal data is collected (uploaded) from the target system. These options are

- **Source:** manual or signal. Selecting manual directs external mode to start logging data when the **Arm trigger** button on the **External Mode Control Panel** is clicked.

Selecting signal tells external mode to start logging data when a selected trigger signal satisfies trigger conditions specified in the **Trigger signal** panel. When the trigger conditions are satisfied (that is, the signal crosses the trigger level in the specified direction) a *trigger event* occurs. If the

trigger is *armed*, external mode monitors for the occurrence of a trigger event. When a trigger event occurs, data logging begins.

- **Arm when connect to target:** If this option is selected, external mode arms the trigger automatically when Simulink connects to the target. If the trigger source is manual, uploading begins immediately. If the trigger mode is signal, monitoring of the trigger signal begins immediately, and uploading begins upon the occurrence of a trigger event.

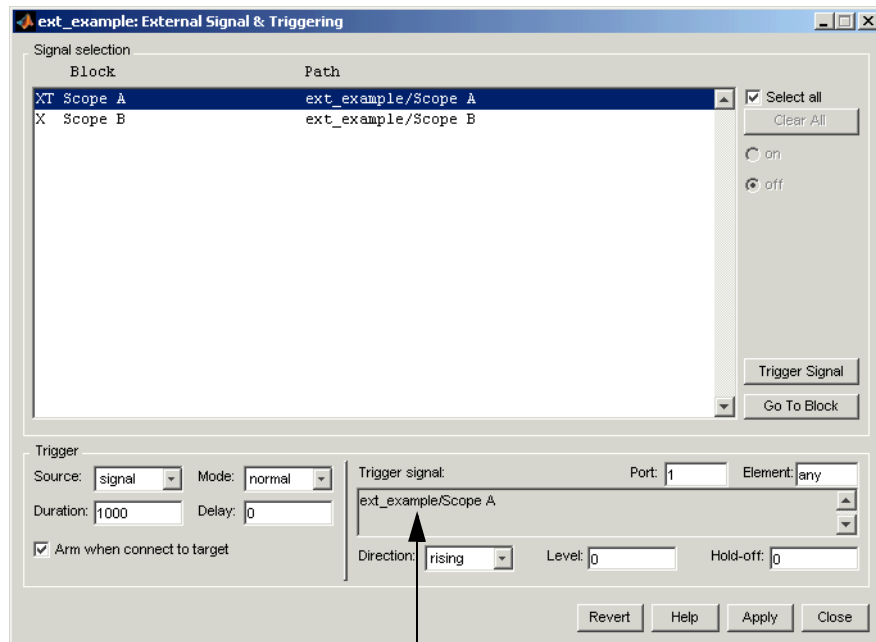
If **Arm when connect to target** is not selected, you must manually arm the trigger by clicking the **Arm trigger** button in the **External Mode Control Panel**.

- **Duration:** The number of base rate steps for which external mode logs data after a trigger event. For example, if the fastest rate in the model is 1 second and a signal sampled at 1 Hz is being logged for a duration of 10 seconds, then external mode will collect 10 samples. If a signal sampled at 2 Hz is logged, only 5 samples will be collected.
- **Mode:** normal or one-shot. In normal mode, external mode automatically rearms the trigger after each trigger event. In one-shot mode, external mode collects only one buffer of data each time you arm the trigger. See “Data Archiving” on page 6-18 for further details on the effect of the **Mode** setting.
- **Delay:** The delay represents the amount of time that elapses between a trigger occurrence and the start of data collection. The delay is expressed in base rate steps, and can be positive or negative. A negative delay corresponds to pretriggering. When the delay is negative, data from the time preceding the trigger is collected and uploaded.

Trigger Signal Selection

You can designate one signal as a trigger signal. To select a trigger signal, select signal from the **Trigger Source** menu. This activates the **Trigger signal** panel (see the figure below). Then, click the desired entry in the **Signal selection** list and click the **Trigger signal** button.

When a signal is selected as a trigger, a T appears to the left of the block’s name in the **Signal selection** list. In the following figure, the Scope A signal is the trigger. Scope B is also selected for viewing, as indicated by the X to the left of the block name.



The Trigger Signal panel

External Signal & Triggering Window with Trigger Selected

After selecting the trigger signal, you can define the trigger conditions in the **Trigger signal** panel, and set the **Port** and **Element** fields located on the right side of the **Trigger** panel.

Setting Trigger Conditions

Note The **Trigger signal** panel and the **Port** and **Element** fields of the **External Signal & Triggering** dialog box are enabled only when trigger **Source** is set to signal.

By default, any element of the first input port of the specified trigger block can cause the trigger to fire (i.e., Port 1, any element). You can modify this behavior

by adjusting the **Port** and **Element** fields located on the right side of the Trigger signal panel. The **Port** field accepts a number or the keyword `last`. The **Element** field accepts a number or the keywords `any` and `last`.

The **Trigger Signal** panel defines the conditions under which a trigger event will occur.

- **Level:** Specifies a threshold value. The trigger signal must cross this value in a designated direction to fire the trigger. By default, the level is 0.
- **Direction:** `rising`, `falling`, or `either`. This specifies the direction in which the signal must be traveling when it crosses the threshold value. The default is `rising`.
- **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.

Data Archiving

Clicking the **Data Archiving** button of the **External Mode Control Panel** opens the **External Data Archiving** dialog box, which supports the following features:

Directory Notes. Use this option to add annotations that pertain to a collection of related data files in a directory.

Clicking the **Edit directory note** button opens the MATLAB editor. Place comments that you want saved to a file in the specified directory in this window. By default, the comments are saved to the directory last written to by data archiving.

File Notes. Clicking **Edit file note** opens a file finder window that is, by default, set to the last file to which you have written. Selecting any MAT-file opens an edit window. Add or edit comments in this window that you want saved with your individual MAT-file.

Data Archiving. Clicking the **Enable Archiving** check box activates the automated data archiving features of external mode. To understand how the archiving features work, it is necessary to consider the handling of data when archiving is not enabled. There are two cases, one-shot and normal mode.

In one-shot mode, after a trigger event occurs, each selected block writes its data to the workspace just as it would at the end of a simulation. If another one-shot is triggered, the existing workspace data is overwritten.

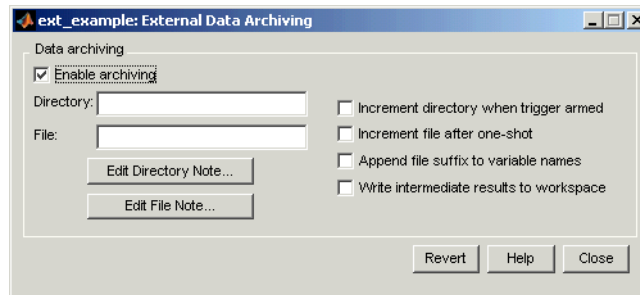
In normal mode, external mode automatically rearms the trigger after each trigger event. Consequently, you can think of normal mode as a series of one-shots. Each one-shot in this series, except for the last, is referred to as an *intermediate result*. Since the trigger can fire at any time, writing intermediate results to the workspace generally results 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 sufficient time exists between triggers for inspection of the intermediate results, then you can override the default behavior by checking the **Write intermediate results to workspace** check box. Note that this option does not protect the workspace data from being overwritten by subsequent triggers.

The options in the **External Data Archiving** dialog box support automatic writing of logging results, including intermediate results, to disk. Data archiving provides the following settings:

- **Directory:** Specifies the directory in which data is saved. External mode appends a suffix if you select **Increment directory when trigger armed**.
- **File:** Specifies the filename in which data is saved. External mode appends a suffix if you select **Increment file after one-shot**.
- **Increment directory when trigger armed:** External mode uses a different directory for writing log files each time that you click the **Arm trigger** button. The directories 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`, etc. Note that this happens automatically in normal mode.
- **Append file suffix to variable names:** Whenever external mode increments filenames, each file contains variables with identical names. Choosing **Append file suffix to variable name** results in each file containing unique variable names. For example, external mode will save a variable named `xdata` in incremental files (`file_1`, `file_2`, etc.) as `xdata_1`, `xdata_2`, and so on. This is useful if you want to load the MAT-files into the workspace and

compare variables in MATLAB. Without the unique names, each instance of xdata would overwrite the previous one in the MATLAB workspace.

This picture shows the **External Data Archiving** dialog box with archiving enabled.



Unless you select **Enable archiving**, entries for the **Directory** and **File** fields are not accepted.

Parameter Downloading

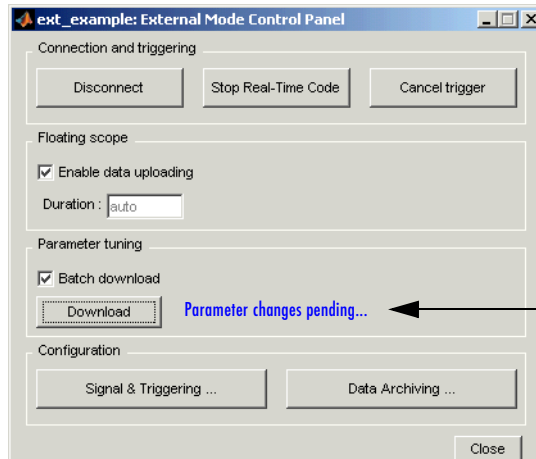
The **batch download** check box on the **External Mode Control Panel** enables or disables batch parameter changes.

By default, **batch download** is not enabled. When **batch download** is not enabled, changes made directly to block parameters are sent immediately to the target. Changes to MATLAB workspace variables are sent when an **Update diagram** is performed.

When **batch download** is enabled, the **Download** button is enabled. Changes made to block parameters are stored locally until you click the **Download** button. When you click the **Download** button, the changes are sent in a single transmission.

When parameter changes have been made and are awaiting batch download, the **External Mode Control Panel** displays the message **Parameter changes pending...** to the right of the download button. (See the figure below.) This message disappears after Simulink receives notification from the target that the new parameters have been installed in the parameter vector of the target system.

The **External Mode Control Panel** with the batch download option activated is shown below.



Parameter changes pending... message appears if unsent parameter value changes are awaiting download.

External Mode Control Panel in Batch Download Mode

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:

- Scope and Floating Scope blocks
- Blocks in the Dials & Gauges Blockset
- 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/simstruc.h`.

- XY Graph blocks

In addition to these types of 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 6-22 for further information.

External mode compatible blocks and subsystems are selected, and the trigger is armed, via the **External Signal & Triggering** dialog box. By default, all 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, generating no code in the target system. Signal Viewing Subsystems run in all simulation modes — normal, accelerated, and external.

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 Select the **Treat as atomic unit** option in the **Block Parameters** dialog box.

See “Nonvirtual Subsystem Code Generation” on page 4-2 for further information on atomic subsystems.

- 2 Use the following `set_param` command to turn the `SimViewingDevice` property on,

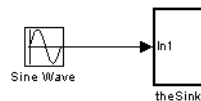
```
set_param('blockname', 'SimViewingDevice', 'on')
```

where 'blockname' is the name of the subsystem.

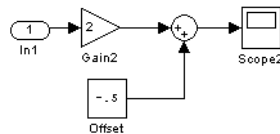
- 3 Make sure the subsystem meets the following requirements:

- It must be a pure sink block. That is, it must contain no Output blocks or Data Store blocks. It can contain Goto blocks only if the corresponding From blocks are contained within the subsystem boundaries.
- It must have no continuous states.

The model shown below, `sink_examp`, contains an atomic subsystem, `theSink`.



The subsystem `theSink`, shown below, 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 their effect upon the uploaded signal.

Note that 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 Simulink after being processed by these blocks, and viewed on sink_exam/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.

External Mode Communications Overview

This section describes how Simulink and the target program communicate, and how and when they transmit parameter updates and signal data to each other.

Depending on the setting of the **Inline parameters** option when the target program is generated, there are differences in the way parameter updates are handled. “The Download Mechanism” on page 6-25 describes the operation of external mode communications with **Inline parameters** off. “Inlined and Tunable Parameters” on page 6-26 describes the operation of external mode with **Inline parameters** on.

The Download Mechanism

In external mode, Simulink does not simulate the system represented by the block diagram. By default, when external mode is enabled, Simulink downloads current values of all parameters to the target system. After the initial download, Simulink remains in a waiting mode until you change parameters in the block diagram or until Simulink receives data from the target.

When you change a parameter in the block diagram, Simulink calls the external interface MEX-file, passing new parameter values (along with other information) as arguments.

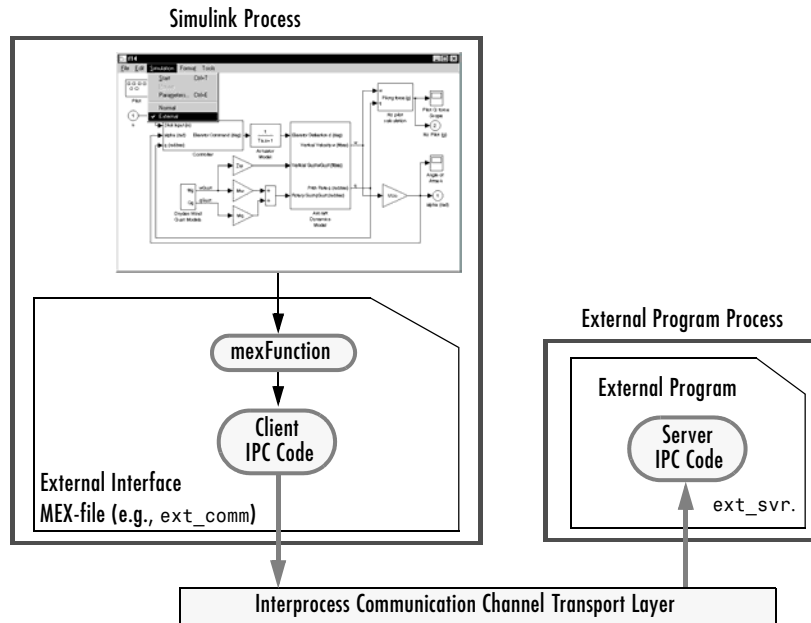
The external interface MEX-file contains code that implements one side of the interprocess communication (IPC) channel. This channel connects the Simulink process (where the MEX-file executes) to the process that is executing the external program. The MEX-file transfers the new parameter values via this 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 following diagram illustrates this relationship.

Simulink 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 via the communication channel.



External Mode Architecture

Inlined and Tunable Parameters

By default, all parameters (except those listed in “Limitations of External Mode” on page 6-35) in an external mode program are tunable; that is, you can change them via the download mechanism described in this section.

If you select the **Inline parameters** option (on the **Optimization** pane of the **Configuration Parameters** dialog box), Real-Time Workshop 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.

Real-Time Workshop 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 the **Model Parameter Configuration** dialog to remove individual parameters from inlining and declare them to be tunable. In addition, the **Model Parameter Configuration** dialog offers you options for controlling how parameters are represented in the generated code.

For more information on tunable parameters, see “Parameters: Storage, Interfacing, and Tuning” on page 5-2.

Automatic Parameter Uploading on Host/Target Connection

Each time Simulink connects to a target program that was generated with **Inline parameters** on, the target program uploads the current value of its tunable parameters (if any) to the host. These values are assigned to the corresponding MATLAB workspace variables. This procedure ensures that the host and target are synchronized with respect to parameter values.

All 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.

Note that automatic parameter uploading takes place only if the target program was generated with **Inline parameters** on. “The Download Mechanism” on page 6-25 describes the operation of external mode communications with **Inline parameters** off.

Client/Server Implementations

Real-Time Workshop provides code to implement both the client and server side using either TCP/IP or serial protocols. You can use the socket-based external mode implementation provided by Real-Time Workshop 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 Simulink 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.

See “Target Interfacing” on page 6-12 for information on selecting a transport layer.

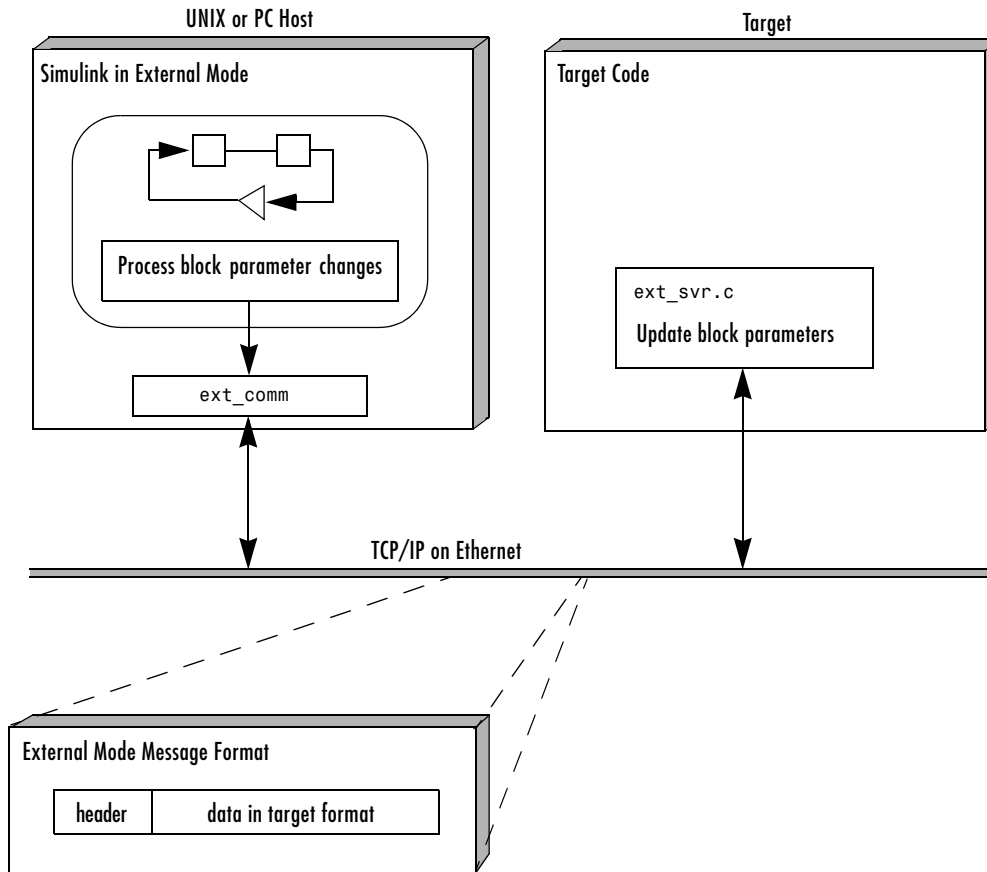
Using the TCP/IP Implementation

You can use TCP/IP-based client/server implementation of external mode with real-time programs on a UNIX or PC system. Chapter 13, “Targeting Tornado for Real-Time Applications,” illustrates the use of external mode in the Tornado environment. For help in customizing external mode transport layers, see “Creating an External Mode Communication Channel” on page 17-18.

In order to use Simulink external mode over TCP/IP, you must

- Have the name of the external interface MEX-file correctly specified in the system target file. The name of the interface will appear as uneditable text in the **Host/Target interface** section of the **Interface** pane of the **Configuration Parameters** dialog. The TCP/IP default is `ext_comm`.
- Be sure that the template makefile is configured to link the proper source files for the TCP/IP server code and that it defines the necessary compiler flags when building the generated code.
- Build the external program.
- Run the external program.
- Set Simulink to external mode and connect to the target.

The following 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 comma-delimited arguments that are passed to the MEX-file.

- **Target network name:** the network name of the computer running the external program. By default, this is the computer on which Simulink is running. The name can be
 - String delimited by single quotes, such as 'myPuter'

- IP address delimited by single quotes, such as '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 if necessary.

You must specify these options in order. For example, if you want to specify the verbosity level (the second argument), then you must also specify the target host name (the first argument).

Note that you can specify command-line options to the external program when you launch it. See “Running the External Program” on page 6-31 for more information.

Using the Serial Implementation

You control host/target communications on serial a channel in a similar fashion to doing so for a TCP/IP channel.

In order to use Simulink external mode over a serial channel, you must

- Execute the target and host on a Windows platform.
- Have the name of the external interface MEX-file correctly specified in the system target file. The name of the interface will appear as uneditable text in the **Host/Target interface** section of the **Interface** pane of the **Configuration Parameters** dialog. The serial default is `serial_win32`.
- Be sure that the template makefile is configured to link the proper source files for the serial server code and that it defines the necessary compiler flags when building the generated code.
- Build the external program.
- Run the external program.
- Set Simulink to external mode and connect to the target.

MEX-File Optional Arguments for Serial Transport

You can specify optional comma-delimited arguments that are passed to the MEX-file in the **MEX-file arguments** field of the **Interface** pane of the **Configuration Parameters** dialog. For serial transport, the optional arguments to `ext_serial_win32_comm` are

- **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
- **Serial port ID** (e.g., 1 for COM1, etc.)

If the target program is executing on the same machine as the host and communications 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).

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** (selected from the set 1200, 2400, 4800, 9600, 14400, 19200, 38400, 57600, 115200, with a default baud rate of 57600)

You must specify these options in order. For example, in order to specify the serial port ID (the second argument), then you must also specify the verbosity level (the first argument).

Note that you can specify command-line options to the external program when you launch it. The following section provides details on using command-line arguments.

Running the External Program

The external program must be running before you can use Simulink in external mode. To run the external program, you type a command of the form

```
model -opt1 ... -optN
```

where *model* is the name of the external program and *-opt1 ... -optN* are options. (See “Command-Line Options for the External Program” on page 6–33.) In the examples in this section, the name of the external program is assumed to be `ext_example`.

Running the External Program Under Windows

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 any options, as in the following example:

```
ext_example -tf inf -w
```

- Alternatively, you can launch the target executable from the MATLAB command prompt. In this case the command must be preceded by an exclamation point (!) and followed by an ampersand (&), as in the following example:

```
!ext_example -tf inf -w &
```

Note that 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 will be unable to communicate commands to MATLAB or manually terminate the executable.

Running the External Program Under UNIX

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 any options, as in the following example:

```
ext_example -tf inf -w
```

- Alternatively, you can launch the target executable from the MATLAB command prompt. In the UNIX environment, if you start the external program from MATLAB, you must run it in the background so that you can still access Simulink. The command must be preceded by an exclamation point (!) and followed by an ampersand (&), as in the following example:

```
!ext_example -tf inf -w &
```

runs the executable from MATLAB by spawning another process to run it.

Command-Line Options for the External Program

External mode target executables generated by Real-Time Workshop support the following command-line options:

- **-tf n** option

The **-tf** option overrides the stop time set for the model in Simulink. 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 will run until the target program receives a stop message from Simulink.

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. See “Using External Mode with the ERT Target” in the Real-Time Workshop Embedded Coder documentation.

Note The **-tf** option works with GRT, GRT malloc, ERT, RSim, and Tornado targets. If you are creating a custom target and want to support the **-tf** option, you must implement the option yourself. See “Creating an External Mode Communication Channel” on page 17-18 for further information.

- **-w** option

The **-w** option 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` option

the `-port` option specifies the TCP/IP port number, `n`, for the target program. The port number of the target program must match that of the host. The default port number is 17725. The port number must be a value between 256 and 65535.

Note The `-tf`, `-w`, and `-port` options are supported by the TCP/IP and serial transport layer modules shipped with Real-Time Workshop (although `-port` is interpreted differently by each). The `-baud` option is serial only. By default, these modules are linked into external mode target executables. If you are implementing a custom external mode transport layer and want to support these options, you must implement them in your code.

Implementing 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 Real-Time Workshop and create a new external interface MEX-file. This advanced topic is described in detail in “Creating an External Mode Communication Channel” on page 17-18. See `matlabroot/rtw/c/src/ext_transport.c` for example code.

Limitations of External Mode

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 any block
- The sample time or the number of sample times
- The integration algorithm for continuous systems
- The name of the model or of any block
- The parameters to the Fcn block

If you cause any of these changes to the block diagram, then you must rebuild the program with newly generated code.

However, parameters in transfer function and state space representation blocks *can* be changed 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 (i.e., 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 blocks, 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.

Error Conditions

If the Simulink block diagram does not match the external program, Simulink displays an error informing you that the checksums do not match (i.e., the model has changed since you generated code). This means that you must rebuild the program from the new block diagram (or reload the correct one) in order to use external mode.

If the external program is not running, Simulink displays an error informing you that it cannot connect to the external program.

Program Architecture

Code is generated by Real-Time Workshop in two styles, depending whether a target is embedded or not. In addition, the structure of code is affected by whether a multitasking environment is available for execution, and on what system and applications modules must be incorporated.

Introduction (p. 7-2)

Code styles and targets appropriate for development of rapid prototyping and embedded systems

Model Execution (p. 7-4)

How code generated from models executes, including single-tasking and multitasking execution, timing, data structures, entry points, and differences between rapid prototyping and embedded code

Rapid Prototyping Program Framework (p. 7-23)

Overall architecture and individual components of programs generated by rapid prototyping targets

Embedded Program Framework (p. 7-36)

Overview of the architecture of programs generated by the Real-Time Workshop Embedded Coder

For a detailed discussion of the structure of embedded real-time code, see the Real-Time Workshop Embedded Coder documentation.

Introduction

Real-Time Workshop generates two styles of code. One code style is suitable for rapid prototyping (and simulation via code generation). The other style is suitable for embedded applications. This chapter discusses the program architecture, that is, the structure of code generated by Real-Time Workshop, associated with these two styles of code. The table below classifies the targets shipped with Real-Time Workshop. For related details about code style and target characteristics, see “Choosing a Code Format for Your Application” on page 3-9.

Code Styles Listed by Target

Target	Code Style (using C unless noted)
Real-Time Workshop Embedded Coder embedded real-time (ERT) target	Embedded — Useful as a starting point when using the generated C code in an embedded application (often referred to as a <i>production code target</i>).
Real-Time Workshop Generic real-time (GRT) target	Rapid prototyping — Use as a starting point for creating a rapid prototyping target that does not use real-time operating system tasking primitives, and for verifying the generated code on your workstation. Uses components of ERT, with a different calling interface.
Real-time malloc target	Rapid prototyping — Similar to the generic real-time (GRT) target except that this target allocates all model working memory dynamically rather than statically declaring it in advance.
Rapid simulation target (RSim)	Rapid prototyping — Non-real-time simulation of your model on your workstation. Useful as a high-speed or batch simulation tool.

Code Styles Listed by Target (Continued)

Target	Code Style (using C unless noted)
S-function target	Rapid prototyping — Creates a C-MEX S-function for simulation of your model within another Simulink model; useful for intellectual property protection.
Tornado (VxWorks) real-time target	Rapid prototyping — Runs model in real time using the VxWorks real-time operating system tasking primitives. Also useful as a starting point for targeting a real-time operating system.
Real-Time Windows target	Rapid prototyping — Runs model in realtime at interrupt level while your PC is running Microsoft Windows in the background.
xPC target	Rapid prototyping — Runs model in real time on target PC running xPC kernel.

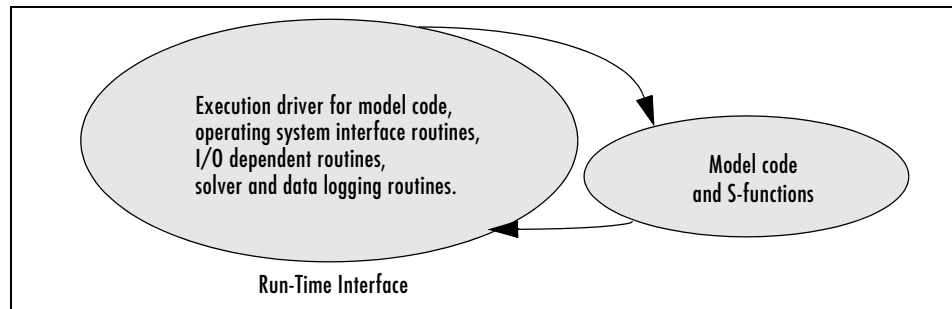
Third-party vendors supply additional targets for Real-Time Workshop. Generally, these can be classified as rapid prototyping targets. For more information about third-party products, see the MathWorks Connections Program Web page: <http://www.mathworks.com/products/connections>.

This chapter is divided into three sections. The first section discusses model execution, the second section discusses the rapid prototyping style of code, and the third section discusses the embedded style of code.

Model Execution

Before looking at the two styles of generated code, you need to have a high-level understanding of how the generated model code is executed. Real-Time Workshop generates algorithmic code as defined by your model. You can include your own code in your model via S-functions. S-functions can range from high-level signal manipulation algorithms to low-level device drivers.

Real-Time Workshop also provides a run-time interface that executes the generated model code. The run-time interface and model code are compiled together to create the model executable. The diagram below 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 realtime. For most models, the multitasking environment will provide the most efficient model execution (i.e., fastest sample rate).

The following concepts are useful in describing how models execute. Function names used in ERT and GRT targets are shown, followed by the comparable GRT-compatible calls in parentheses.

- **Initialization:** `mdl_initialize` (`MdlInitializeSizes`, `MdlInitializeSampleTimes`, `MdlStart`) initializes the run-time interface code and the model code.

- **ModelOutputs:** Calling all blocks in your model that have a sample hit at the current time and having them produce their output. *model_outputs* (MdlOutputs) 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 run-time interface integrates the derivatives to update the continuous states.
- **ModelUpdate:** *model_update* (MdlUpdate) calls all 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:** Calling all blocks in your model that have continuous states and having them update their derivatives. *model_derivatives* (MdlDerivatives) is only called in minor time steps.
- **ModelTerminate:** *model_terminate* (MdlTerminate) 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.

The identifying names in the preceding list (ModelOutputs, etc.) identify functions in pseudocode examples shown in the following topics.

- “Models for Non-Real-Time Single-Tasking Systems” on page 7-5
- “Models for Non-Real-Time Multitasking Systems” on page 7-6
- “Models for Real-Time Single-Tasking Systems” on page 7-7
- “Models for Real-Time Multitasking Systems” on page 7-9
- “Models for Multitasking Systems that Use Real-Time Tasking Primitives” on page 7-11

For a complete set of correspondences between GRT and ERT function identifiers, see the table “Identifiers for Real-Time Model Data Structure Variants” on page 7-31.

Models for 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 outputs.
```

```
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 any 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.

Models for 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
                       -- outputs.
    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 there were no preemption in a real-time system.

Note that multitasking execution assumes that all tasks are multiples of the base rate. Simulink 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`.

Models for 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 outputs.
    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 all rates run at the same priority as the fastest rate (base rate).

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 these steps must occur before the next interrupt.

Models for 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 outputs.
  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
                        -- outputs
    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
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.

Models for Multitasking Systems that Use 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 outputs.
    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".

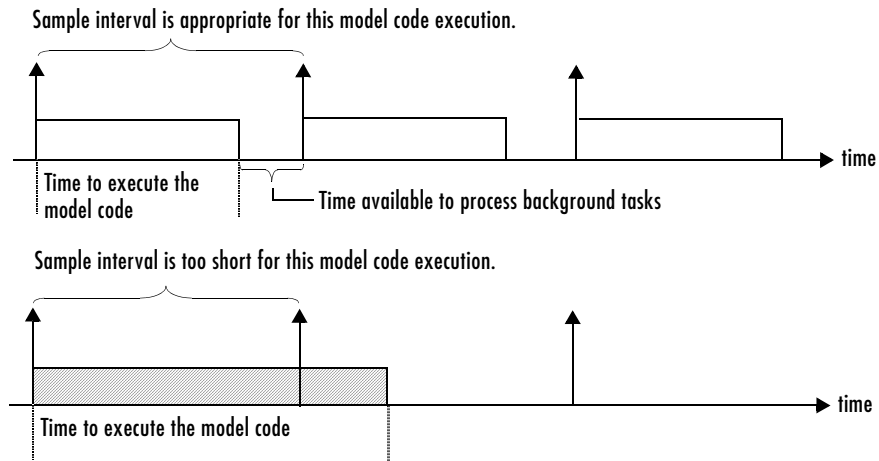
    Start clock that does a "semGive" on a clockSem semaphore.
    Wait on "model-running" semaphore.
    Shutdown
}
```

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.

Program Timing

Real-time programs require careful timing of the task invocations (either via an interrupt or a real-time operating system tasking primitive) to ensure 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 following diagram 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 (i.e., the final time is 0 or infinite so the `while` loop never exits), then the shutdown code never executes.

For more information on how the timing engine works, see Chapter 15, "Timing Services."

Program Execution

As the previous section indicates, 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 at the appropriate time to ensure real-time execution of the model code.

The way the program manages tasks depends on capabilities of the environment in which it operates.

External Mode Communication

External mode allows communication between the Simulink block diagram and the stand-alone 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 Simulink.

Data Logging in Single Tasking and Multitasking Model Execution

The Real-Time Workshop data-logging features, described in “Data Import and Export Options” on page 2-13, enable you to 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, all 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 $t = k*10$, $k=0,1,2,\dots$ 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 output 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.

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 Real-Time Workshop Embedded Coder 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 run-time function, *model_step*.

Thus, when you look again at the model execution pseudocode presented earlier in this chapter, you can eliminate the `Loop . . . EndLoop` statements, and

group ModelOutputs, LogTXY, and ModelUpdate into a single statement, *model_step*.

For a detailed discussion of how generated embedded code executes, see the Real-Time Workshop Embedded Coder documentation.

Rapid Prototyping Model Functions

The rapid prototyping code defines the following functions that interface with the run-time interface:

- **Model()**: The model registration function. This function initializes the work areas (e.g., allocating and setting pointers to various data structures) needed 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 run-time interface 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 that has a constant sample time.
- **MdlOutputs(int_T tid)**: `MdlOutputs` updates the output of blocks at appropriate times. 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 run-time interface during major and minor time steps. The major time steps are when the run-time interface is taking an

actual time step (i.e., 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.

- `Md1Update(int_T tid)`: `Md1Update` updates the discrete states and work vector state information (i.e., 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 run-time interface after the major `Md1Outputs` has been executed.
- `Md1Derivatives(void)`: `Md1Derivatives` returns the block derivatives. This routine 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.
- `Md1Terminate(void)`: `Md1Terminate` contains any block shutdown code. `Md1Terminate` is called by the run-time 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 the appropriate section of `Md1Output`.

$$x_{d+1} = f_u(t, x_d, u)$$

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 `Md1Update`.

$$\dot{x} = f_d(t, x_c, u)$$

The derivatives x are a function of the current input. Each block that has continuous states provides its derivatives to the solver (e.g., `ode5`) in `Md1Derivatives`. 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 Output 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 Real-Time Workshop generates.

The example below 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.
     */
}

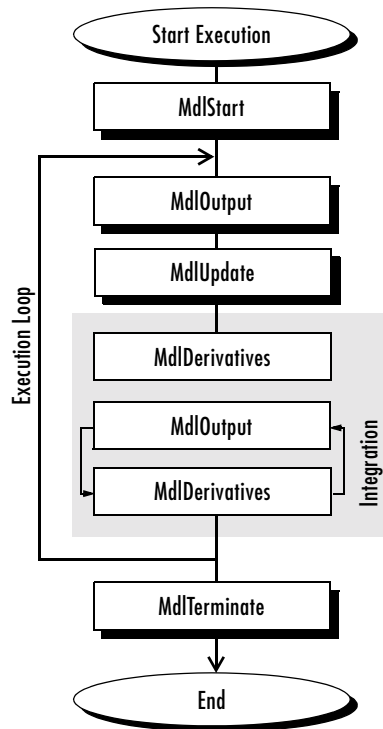
void MdlOutputs(int_T tid)
{
    /* Compute: y = f0(t,xc,xd,u) for each block as needed. */
}

void MdlUpdate(int_T tid)
{
    /* Compute: xd+1 = fu(t,xd,u) for each block as needed. */
}

void MdlDerivatives(void)
{
    /* Compute: dxc = fd(t,xc,u) for each block as needed. */
}

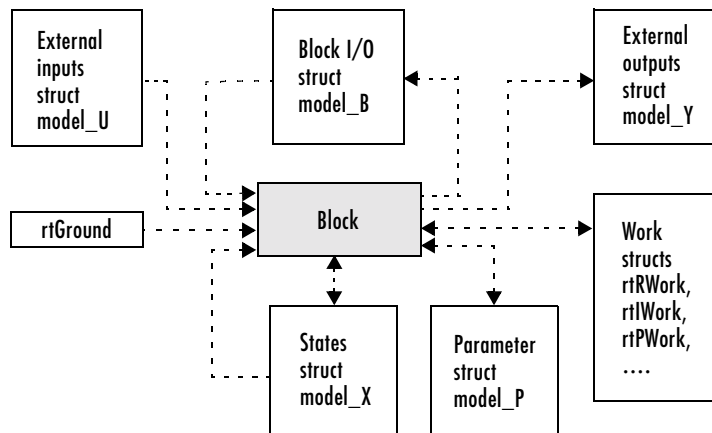
void MdlTerminate(void)
{
    /* Perform shutdown code for any blocks that
     have a termination action */
}
```

The following 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 Md1 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 following 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, Simulink and Real-Time Workshop reduce the size of the *model_B* structure by
 - Reusing the entries in the *model_B* structure
 - Making other entries local variables

See “Signal Storage, Optimization, and Interfacing” on page 5-19 for further 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 any 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 all block parameters that can be changed during execution (e.g., 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.

Embedded Model Functions

The Real-Time Workshop Embedded Coder target generates the following functions:

- *model_initialize*: Performs all model initialization and should be called once before you start executing your model.
- If the **Single output/update function** code generation option is selected, then you see
 - *model_step*(int_T tid): Contains the output and update code for all blocks in your model.

Otherwise you see

- *model_output*(int_T tid): Contains the output code for all blocks in your model.
 - *model_update*(int_T tid): This contains the update code for all blocks in your model.
- If the **Terminate function required** code generation option is selected, then you see
 - *model_terminate*: This contains all model shutdown code and should be called as part of system shutdown.

See the Real-Time Workshop Embedded Coder documentation for complete descriptions of these functions in the context of the Real-Time Workshop Embedded Coder.

Rapid Prototyping Program Framework

The code modules generated from a Simulink model — *model.c*, *model.h*, and other files — implement the model's system equations, contain block parameters, and perform initialization.

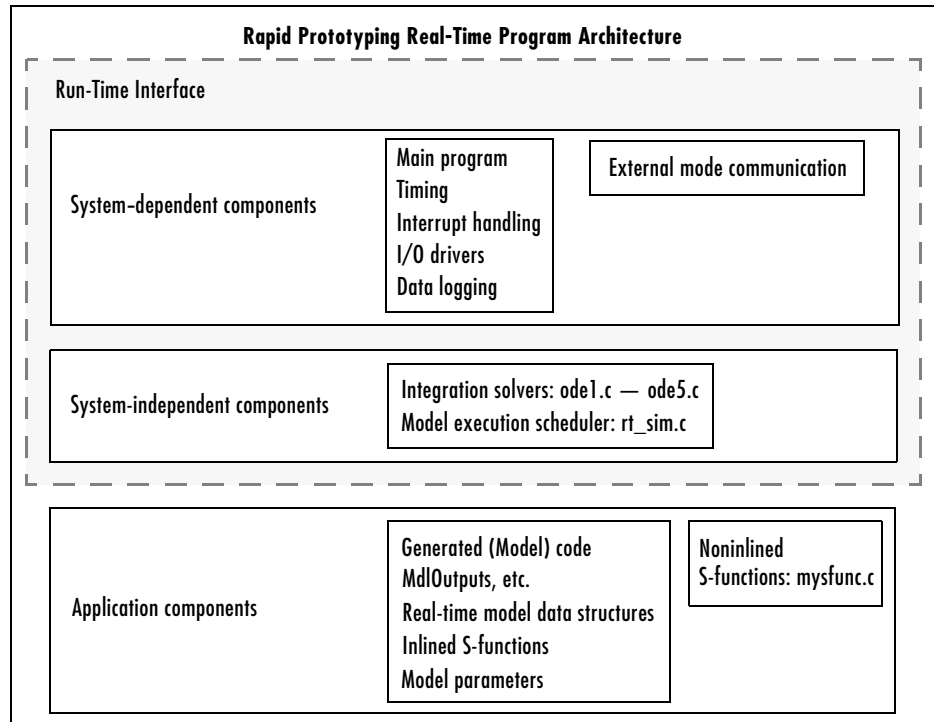
The Real-Time Workshop program framework provides the additional source code necessary to build the model code into a complete, stand-alone program. The program framework consists of *application modules* (files containing source code to implement required functions) designed for a number of different programming environments.

The automatic program builder ensures that the program is created with the proper modules once you have configured your template makefile. The application modules and the code generated for a Simulink model are implemented using a common API. This API defines a data structure (called a *real-time model*, sometimes abbreviated as *rtM*) that encapsulates all data for your model.

This API is similar to that of S-functions, with one major exception: the API assumes that there is only one instance of the model, whereas S-functions can have multiple instances. The function prototypes also differ from S-functions.

Rapid Prototyping Program Architecture

The structure of a real-time program consists of three components. Each component has a dependency on a different part of the environment in which the program executes. The following diagram illustrates this structure.



Rapid Prototyping Program Architecture

The Real-Time Workshop architecture consists of three parts, the first two of which include system-dependent components and system-independent components. Together these two parts form the *run-time interface*.

This architecture adapts readily to a wide variety of environments by isolating the dependencies of each program component. The following sections discuss each component in more detail and include descriptions of the application modules that implement the functions carried out by the system-dependent components, system-independent components, and application components.

Rapid Prototyping System-Dependent Components

These components contain the program's main function, which controls program timing, creates tasks, installs interrupt handlers, enables data logging, and performs error checking.

The way in which application modules implement these operations depends on the type of computer. This means that, for example, the components used for a PC-based program perform the same operations, but differ in method of implementation from components designed to run on a VME target.

The main Function

The main function in a C program is the point where execution begins. In Real-Time Workshop application programs, the main function must perform certain operations. These operations can be grouped into three categories: initialization, model execution, and program termination.

Initialization

- Initialize special numeric parameters `rtInf`, `rtMinusInf`, and `rtNaN`. These are variables that the model code can use.
- Call the model registration function to get a pointer to the real-time model. The model registration function has the same name as your model. It is responsible for initializing real-time model fields and any S-functions in your model.
- Initialize the model size information in the real-time model. This is done by calling `MdlInitializeSizes`.
- Initialize a vector of sample times and offsets (for systems with multiple sample rates). This is done by calling `MdlInitializeSampleTimes`.
- Get the model ready for execution by calling `MdlStart`, which initializes states and similar items.
- Set up the timer to control execution of the model.
- Define background tasks and enable data logging, if selected.

Model Execution

- Execute a background task: for example, communicate with the host during external mode simulation or introduce a wait state until the next sample interval.
- Execute model (initiated by interrupt).
- Log data to buffer (if data logging is used).
- Return from interrupt.

Program Termination

- Call a function to terminate the program if it is designed to run for a finite time — destroy the real-time model data structure, deallocate memory, and write data to a file.

Rapid Prototyping Application Modules for System-Dependent Components

The application modules contained in the system-dependent components generally include a main module such as `rt_main.c`, containing the main entry point for C. There can also be additional application modules for such things as I/O support and timer handling.

Rapid Prototyping System-Independent Components

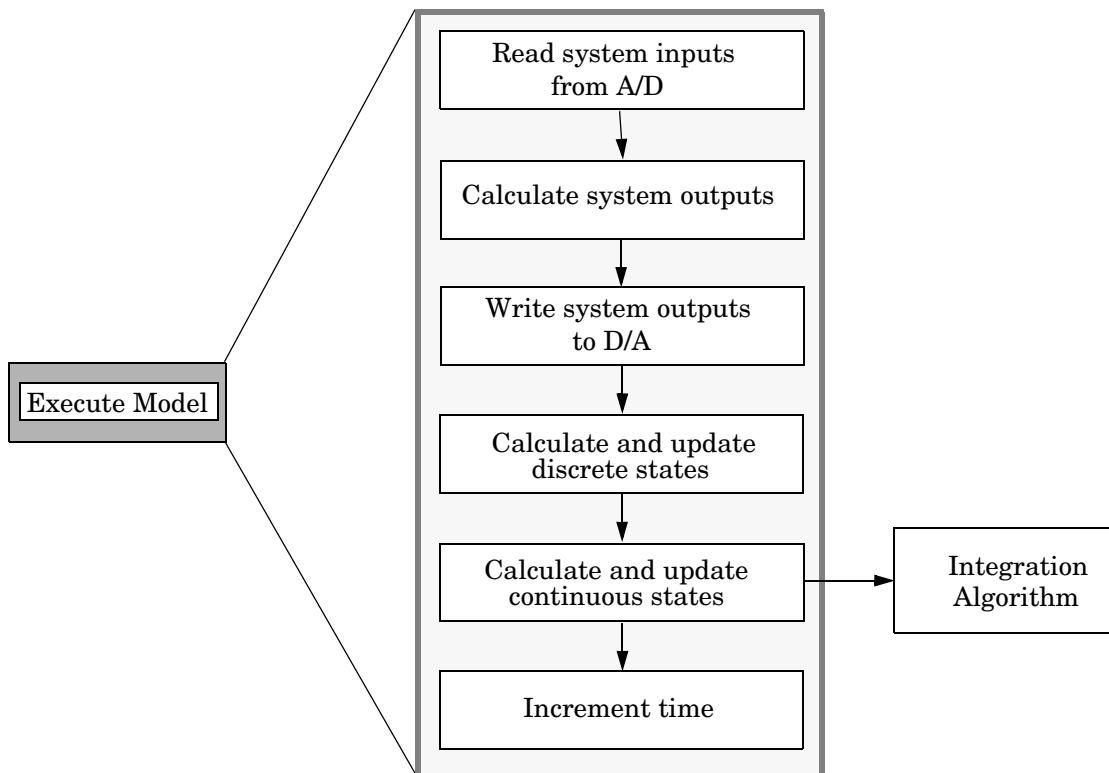
These components are collectively called system independent because all environments use the same application modules to implement these operations. This section steps through the model code (and if the model has continuous states, calls one of the numerical integration routines). This section also includes the code that defines, creates, and destroys the real-time model data structure (`rtM`). The model code and all S-functions included in the program define their own `SimStructs`.

The model code execution driver calls the functions in the model code to compute the model outputs, update the discrete states, integrate the continuous states (if applicable), and update time. These functions then write their calculated data to the real-time model.

Model Execution

At each sample interval, the main program passes control to the model execution function, which executes one step through the model. This step reads inputs from the external hardware, calculates the model outputs, writes outputs to the external hardware, and then updates the states.

The following diagram illustrates these steps.



Executing the Model

Note that this scheme writes the system outputs to the hardware before the states are updated. Separating the state update from the output calculation minimizes the time between the input and output operations.

Integration of Continuous States

The real-time program calculates the next values for the continuous states based on the derivative vector, dx/dt , for the current values of the inputs and the state vector.

These derivatives are then used to calculate the next values of the states using a state-update equation. This is the state-update equation for the first-order Euler method (ode1)

$$x = x + \frac{dx}{dt}h$$

where h is the step size of the simulation, x represents the state vector, and dx/dt is the vector of derivatives. Other algorithms can make several calls to the output and derivative routines to produce more accurate estimates.

Note, however, that real-time programs use a fixed-step size because it is necessary to guarantee the completion of all tasks within a given amount of time. This means that, while you should use higher order integration methods for models with widely varying dynamics, the higher order methods require additional computation time. In turn, the additional computation time might force you to use a larger step size, which can diminish the improvement of accuracy initially sought from the higher order integration method.

Generally, the stiffer the equations, (i.e., the more dynamics in the system with widely varying time constants), the higher the order of the method that you must use.

In practice, the simulation of very stiff equations is impractical for real-time purposes except at very low sample rates. You should test fixed-step size integration in Simulink to check stability and accuracy before implementing the model for use in real-time programs.

For linear systems, it is more practical to convert the model that you are simulating to a discrete time version, for instance, using the `c2d` function in the Control System Toolbox.

Application Modules for System-Independent Components

The system-independent components include these modules:

- `ode1.c`, `ode2.c`, `ode3.c`, `ode4.c`, `ode5.c`: These modules implement the integration algorithms supported for real-time applications. See the Simulink documentation for more information about these fixed-step solvers.
- `rt_sim.c`: Performs the activities necessary for one time step of the model. It calls the model function to calculate system outputs and then updates the discrete and continuous states.
- `simstruc_types.h`: Contains definitions of various events, including subsystem enable/disable and zero crossings. It also defines data-logging variables.

The system-independent components also include code that defines, creates, and destroys the real-time model data structure. All S-functions included in the program define their own `SimStructs`.

The `SimStruct` data structure encapsulates all the data relating to an S-function, including block parameters and outputs. See the [Writing S-Functions](#) documentation for more information about the `SimStruct`.

Rapid Prototyping Application Components

The application components contain the generated code for the Simulink model, including the code for any S-functions in the model. This code is referred to as the model code because these functions implement the Simulink model.

However, the generated code contains more than just functions to execute the model (as described in the previous section). There are also functions to perform initialization, facilitate data access, and complete tasks before program termination. To perform these operations, the generated code must define functions that

- Create the real-time model
- Initialize model size information in the real-time model
- Initialize a vector of sample times and sample time offsets and store this vector in the real-time model
- Store the values of the block initial conditions and program parameters in the real-time model

- Compute the block and system outputs
- Update the discrete state vector
- Compute derivatives for continuous models
- Perform an orderly termination at the end of the program (when the current time equals the final time, if a final time is specified)
- Collect block and scope data for data logging (either with Real-Time Workshop or third-party tools)

The Real-Time Model Data Structure

The real-time model data structure encapsulates model data and associated information necessary to fully describe the model. Its contents include

- Model parameters, inputs, and outputs
- Storage areas, such as `dWork`
- Timing information
- Solver identification
- Data logging information
- Simstructs for all child S-functions
- External mode information

The required information is stored in fields in the real-time model structure, which is defined in `model.h` as

```
/* Real-time Model Data Structure */
struct _rtModel_model_Tag {
    const char *path;
    const char *modelName;
    struct SimStruct_tag * *childSfunctions;
    const char *errorStatus;
    SS_SimMode simMode;
    RTWLogInfo *rtwLogInfo;
    RTWExtModeInfo *extModeInfo;
    RTWSolverInfo solverInfo;
    RTWSolverInfo *solverInfoPtr;
    void *sfcnInfo;

    /*
     * ModelData:
    */
};
```

```

    * The following substructure contains information regarding
    * the data used in the model.
    */
    .
    .
}

```

The (possibly mangled) name of the model replaces *model* in the above tag. The individual substructures have been omitted, as they can vary.

For GRT targets, *model.h* also includes aliases to map global identifiers to identifiers used in prior versions (rtB, rtP, rtY, etc.). The following table lists the structure identifiers used in the generated code for these variants of the real-time model data structure. The column **GRT Symbol** contains the old-style (pre-Version 6) GRT identifiers, which are still used by the GRT calling interface, but not within the generated code.

Identifiers for Real-Time Model Data Structure Variants

Identifier	GRT Symbol	Data
<i>model_B</i>	rtB	Block IO
<i>model_U</i>	rtU	External inputs
<i>model_X</i>	rtX	Continuous states
<i>model_Xdot</i>	rtXdot	State derivatives
<i>model_Xdix</i>	rtXdis	Continuous state disabled
<i>model_Y</i>	rtY	External outputs
<i>model_P</i>	rtP	Parameters
rts	rts	Child Simstruct
<i>model_DWork</i>	rtDWork	DWork
<i>model_ConstB</i>	rtC	Constant block IO define, structure
<i>model_ConstP</i>	rtcP	Constant parameter Structure

Identifiers for Real-Time Model Data Structure Variants

Identifier	GRT Symbol	Data
<i>model_PrevZC</i>	rtPrevZCSigState	Previous zero-crossing signal states
<i>model_NonsampledZC</i>	rtNonsampledZC	Nonsampled zero-crossings

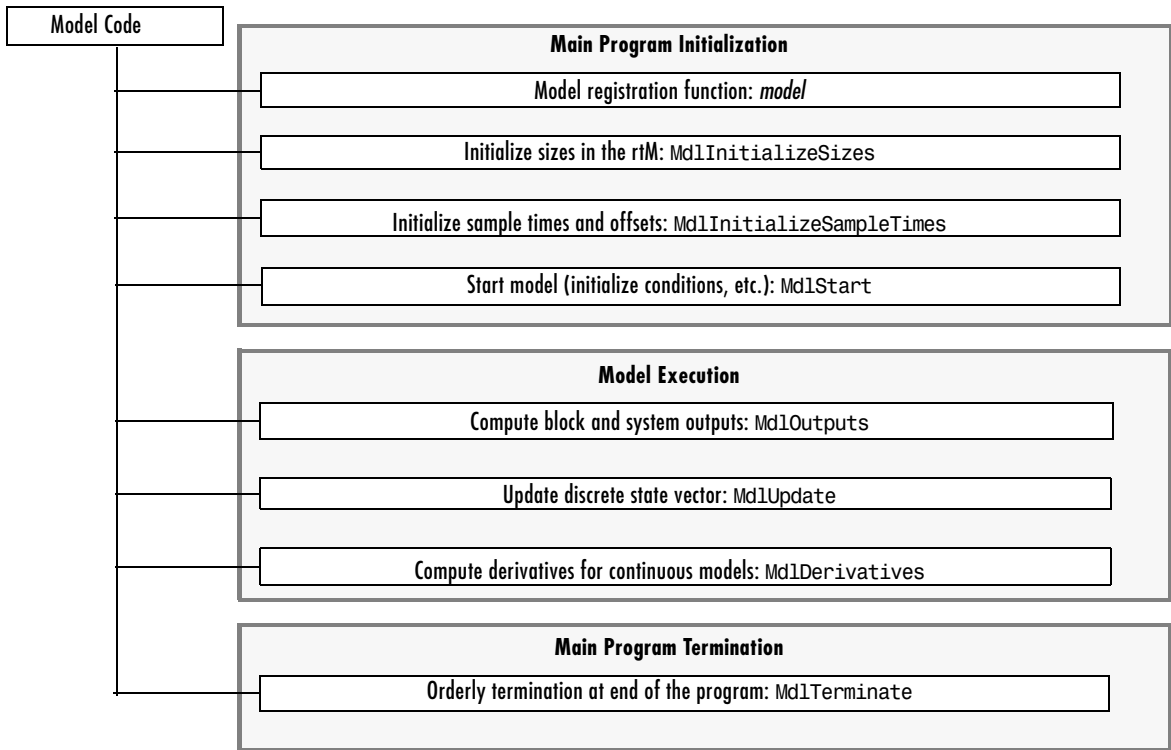
Note that users of Real-Time Workshop Embedded Coder can tailor identifiers, and can make them look like the GRT symbols listed above, should they desire such a coding style. The above GRT-ERT identifier equivalences (or at least as many of them as are required to build a given model) are established via a set of `#define` macros in *model.h*, under the comment `/* Backward compatible GRT Identifiers */`.

The real-time model data structure is used for all targets. Prior to Version 5, the ERT target used the `rtObject` data structure, and other targets used the `Simstruct` data structure for encapsulating model data. Now all targets are treated the same, except for the fact that the real-time model data structure is *pruned* for ERT targets to save space in executables. Even when not pruned, the real-time model data structure is more space efficient than the root `Simstruct` used by earlier releases for non-ERT targets, as it only contains fields for child (S-function) `Simstructs` that are actually used in a model.

Rapid Prototyping Model Code Functions

The functions defined by the model code are called at various stages of program execution (i.e., initialization, model execution, or program termination).

The following diagram illustrates the functions defined in the generated code and shows what part of the program executes each function.



Execution of the Model Code

The Model Registration Function

The model registration function has the same name as the Simulink model from which it is generated. It is called directly by the main program during initialization. Its purpose is to initialize and return a pointer to the real-time model data structure.

Models Containing S-Functions

A *noninlined S-function* is any C MEX S-function that is not implemented using a customized TLC file. If you create a C MEX S-function as part of a Simulink model, it is by default noninlined unless you write your own TLC file that inlines it within the body of the `model.c` code. Real-Time Workshop automatically incorporates your noninlined C code S-functions into the

program if they adhere to the S-function API described in the Simulink documentation.

This format defines functions and a `SimStruct` that are local to the S-function. This allows you to have multiple instances of the S-function in the model. The model's real-time model data structure contains a pointer to each S-function's `SimStruct`.

Code Generation and S-Functions

If a model contains S-functions, the source code for the S-function must be on the search path the make utility uses to find other source files. The directories that are searched are specified in the template makefile that is used to build the program.

S-functions are implemented in a way that is directly analogous to the model code. They contain their own public registration functions (called by the top-level model code) that initialize static function pointers in their `SimStructs`. When the top-level model needs to execute the S-function, it does so via the function pointers in the S-function's `SimStruct`. There can be more than one S-function with the same name in your model. This is accomplished by having function pointers to static functions.

Inlining S-Functions

You can incorporate C MEX S-functions, along with the generated code, into the program executable. You can also write a target file for your C MEX S-function to inline the S-function, thus improving performance by eliminating function calls to the S-function itself. For more information on inlining S-functions, see the Target Language Compiler documentation.

Application Modules for Application Components

When Real-Time Workshop generates code, it produces the following files:

- *model.c*: C code generated from the Simulink block diagram. This code implements the block diagram's system equations as well as performing initialization and updating outputs.
- *model_data.c*: Optional file containing data for parameters and constant block I/O, which are also declared as `extern` in *model.h*. Only generated when *model_P* and *model_ConstB* structures are populated.
- *model_types.h*: Forward declarations for the real-time model data structure and the parameters data structure.
- *model.h*: Header file containing the block diagram's simulation parameters, I/O structures, work structures, and other declarations.
- *model_private.h*: Header file containing declarations of exported signals and parameters.

These files are named for the Simulink model from which they are generated.

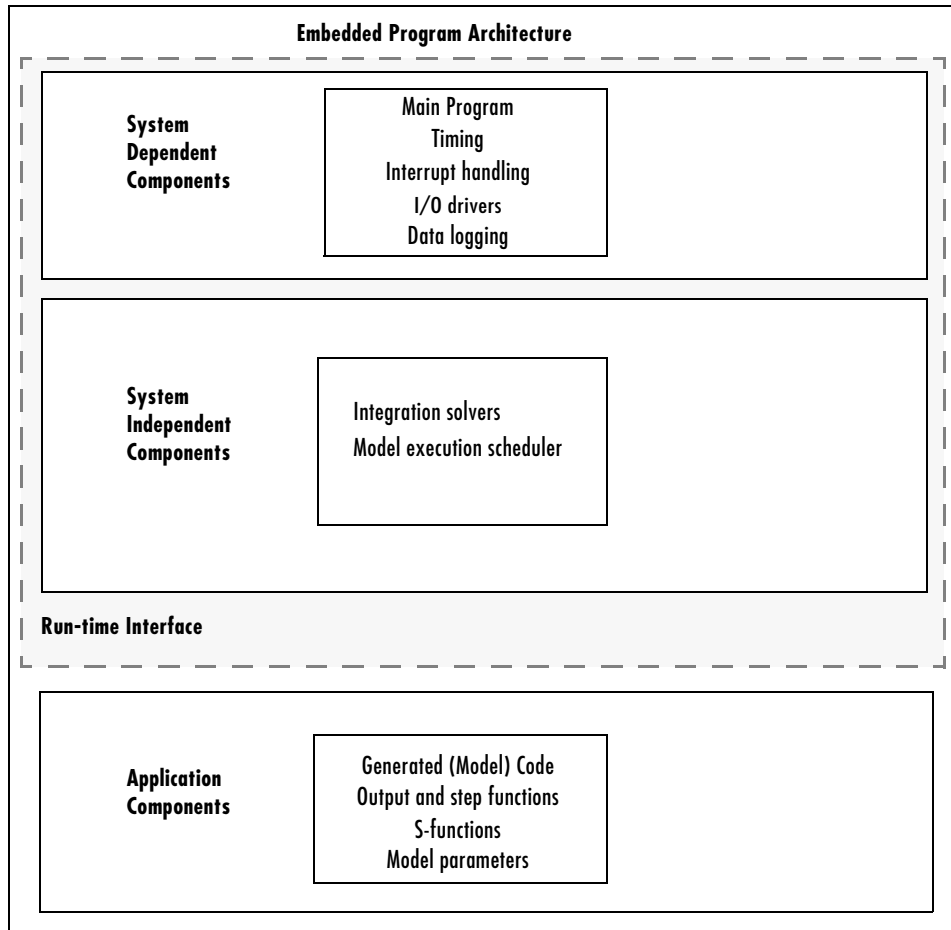
In addition, a dummy include file always named *rtmodel.h* is generated, which includes the above model-specific data structures and entry points. This enables the (static) target-specific main programs to reference files generated by Real-Time Workshop without needing to know the names of the models involved.

Another dummy file, *rtwtypes.h*, is generated, which simply includes *simstruc_types.h* (only for GRT and GRT-malloc targets).

If you have created custom blocks using C MEX S-functions, you need the source code for these S-functions available during the build process.

Embedded Program Framework

The Real-Time Workshop Embedded Coder provides a framework for embedded programs. Its architecture is outlined by the following figure.



Embedded Program Framework Architecture

Note the similarity between this architecture and the rapid prototyping architecture in the figure “Rapid Prototyping Program Architecture” on

page 7-24. The main difference is the use of the `rtModel` data structure in place of the `SimStruct` data structure.

Using the above figure, you can compare the embedded style of generated code, used in the Real-Time Workshop Embedded Coder, with the rapid prototyping style of generated code of the previous section. Most of the rapid prototyping explanations in the previous section hold for the Real-Time Workshop Embedded Coder target. The Real-Time Workshop Embedded Coder target simplifies the process of using the generated code in your custom-embedded applications by providing a model-specific API and eliminating the `SimStruct`. This target contains the same conceptual layering as the rapid prototyping target, but each layer has been simplified.

For a discussion of the structure of embedded real-time code, see the Real-Time Workshop Embedded Coder documentation.

Models with Multiple Sample Rates

The following sections explain and illustrate how Simulink and Real-Time Workshop handle multirate models, depending on whether code is being generated for single-tasking or multitasking environments.

Introduction (p. 8-2)

Describes types of sample times and issues to consider regarding execution of multirate models

Single-tasking vs. Multitasking Environments (p. 8-3)

Discusses how Real-Time Workshop handles execution of multirate systems, in both multitasking and pseudo-multitasking environments

Sample Rate Transitions (p. 8-12)

Shows how to handle transitions between blocks with unequal sample rates using Rate Transition blocks

Single-tasking and Multitasking Execution of a Model: an Example (p. 8-24)

Discusses how an example model executes in both single-tasking and multitasking solver modes, with timing diagrams

Introduction

A Simulink block can be classified, according to how it handles sample time, as *constant*, *continuous-time*, *discrete-time*, *inherited*, or *variable*:

- Constant — Constant, Width blocks
- Continuous-time — Integrator, Derivative, Transfer Function blocks
- Discrete-time — Unit Delay, Digital Filter blocks
- Inherited — Gain, Sum, Lookup Table blocks
- Variable — Blocks that set the time of the next hit based upon current information. Examples include the Pulse Generator and some S-Function blocks. These blocks work only with variable-step solvers.

Blocks in the inherited category assume the sample times of the blocks that are driving them. Continuous blocks have a nominal sample time of zero. Every Simulink block therefore has a sample time, whether it is explicit, as in the case of continuous or discrete blocks, or implicit, as in the case of inherited blocks. S-Function blocks can be designed to operate at different sample times at different input and output ports. S-Function blocks that do this are said to have *port-based sample times*. Built-in blocks—whether constant, continuous-time, discrete-time, inherited, or variable—have *block-based sample times*. For additional information on how S-function blocks handle multiple sample times, see Sample Times in the Simulink S-function documentation.

Simulink allows you to create models without any restrictions on connections between blocks with different sample times. It is therefore possible to have blocks with differing sample times in a model (a mixed-rate system). A possible advantage of employing multiple sample times is improved efficiency when executing in a multitasking real-time environment.

Simulink provides considerable flexibility in building these mixed-rate systems. However, the same flexibility also allows you to construct models for which the code generator cannot generate correct real-time code for execution in a multitasking environment. To make multirate models operate correctly in real time (i.e., to give the right answers), you sometimes must modify your model or instruct Simulink to do it for you. In general, the modifications involve placing Rate Transition blocks between blocks that have unequal sample rates. The sections that follow discuss the issues you must address to use a mixed-rate model successfully in a multitasking environment.

Single-tasking vs. Multitasking Environments

There are two execution modes for a fixed-step Simulink model: single-tasking and multitasking. These modes are available only for fixed-step solvers. You use the **Tasking mode for periodic sample times** pull-down menu on the **Solver** pane of the **Configuration Parameters** dialog box to specify how to execute your model. Auto mode (the default) selects multitasking execution for a mixed-rate model, and otherwise selects single-tasking execution. You can also select Single-tasking or MultiTasking execution explicitly.

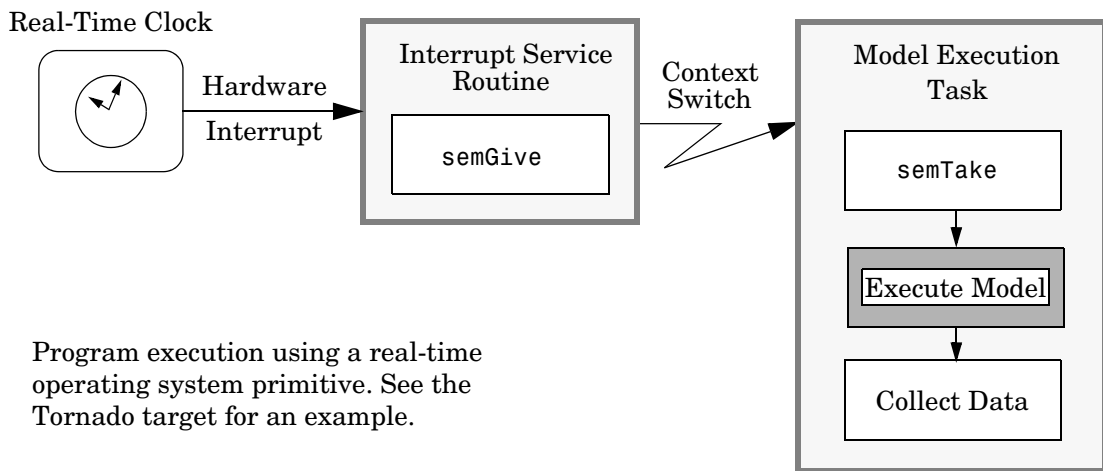
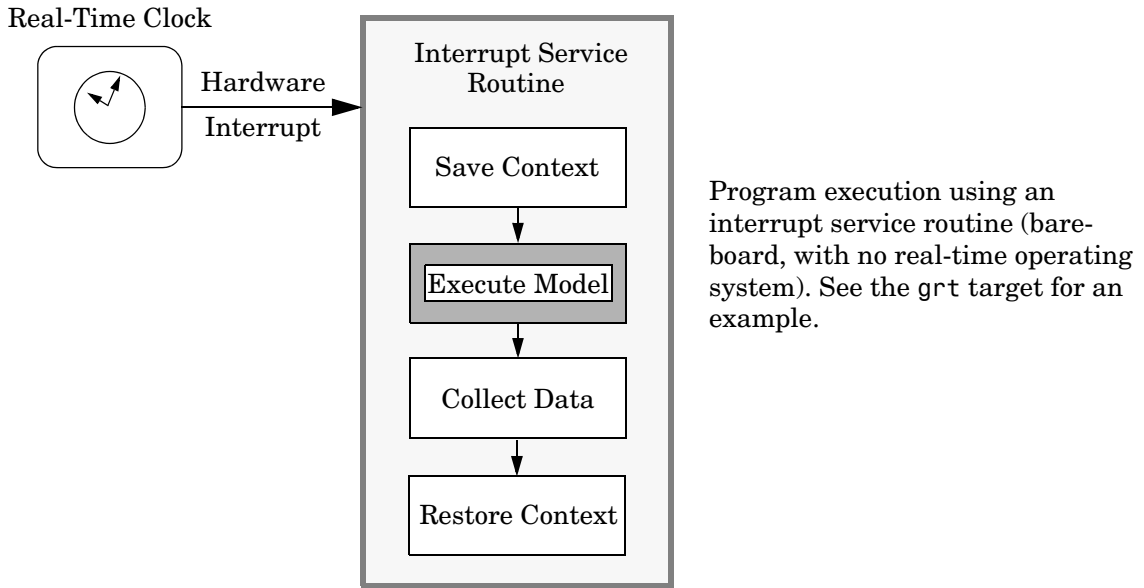
Execution of models in a real-time system can be done with the aid of a real-time operating system, or it can be done on a *bare-board* target, where the model runs in the context of an interrupt service routine (ISR).

Note that the fact that a system (such as UNIX or Microsoft Windows) is multitasking does not guarantee that the program can execute in real time. This is because it is not guaranteed that the program can preempt other processes when required.

In operating systems (such as PC-DOS) where only one process can exist at any 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 following figure illustrates this difference.



Real-Time Program Execution

This chapter focuses on when and how the run-time interface executes your model. See “Program Execution” on page 7-13 for a description of what happens during model execution.

Executing 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 code assigns each block a *task identifier* (tid) to associate the block with the task that executes at the block’s sample rate.

Certain restrictions apply to the sample rates that you can use:

- The sample rate of any block must be an integer multiple of the base (i.e., the fastest) sample period.
- For models containing continuous blocks,
 - When **Periodic sample time constraint** is unconstrained, The base sample period is determined by the **Fixed step size** specified on the **Solver** pane of the **Configuration parameters** dialog box.
 - When **Periodic sample time constraint** is specified, fixed step size is the first element of the sample time matrix that you specify in the companion option **Sample time properties**.

Continuous blocks always execute via an integration algorithm that runs at the base sample rate.

- For purely discrete models,
 - When **Periodic sample time constraint** is unconstrained, the base sample period is determined by the **Fixed step size** specified on the **Solver** pane of the **Configuration parameters** dialog box. When you choose Auto, this is the fastest sample time specified in the model.
 - 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**.

Note that the base sample period is the fastest sample period 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 and Pseudomultitasking

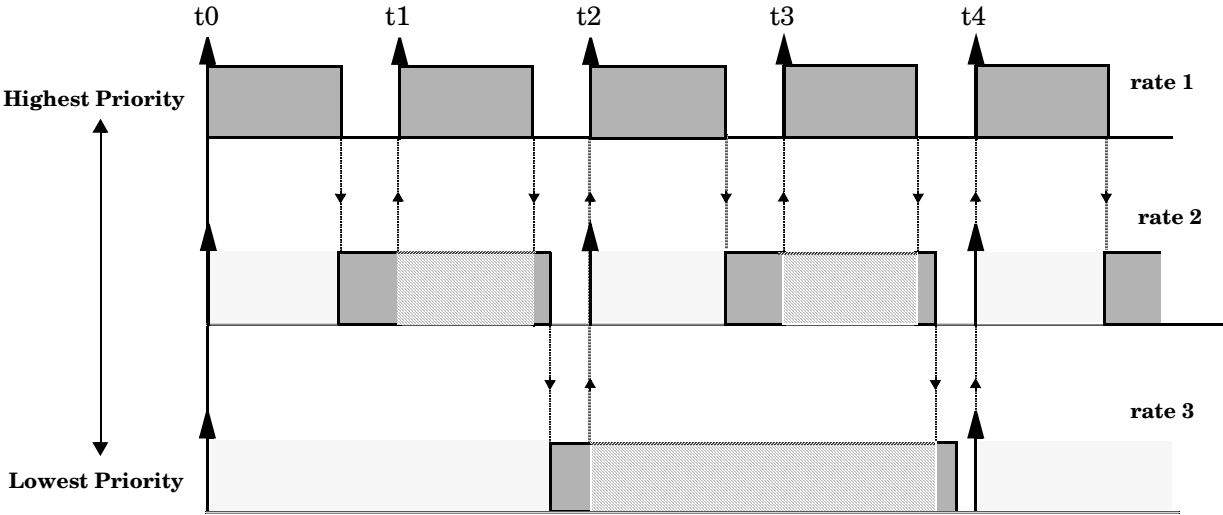
In a multitasking environment, the blocks with the fastest sample rates are executed by the task with the highest priority, the next fastest blocks are executed by a task with the next higher priority, and so on. Time available in between the processing of high-priority tasks is used for processing lower priority tasks. This results in efficient program execution.







See “Multitasking System Execution” on page 8-7 for a graphical representation of task timing.

In multitasking environments (i.e., a real-time operating system), you can define separate tasks and assign them priorities. In a bare-board target (i.e., no real-time operating system present), you cannot create separate tasks. However, Real-Time Workshop application modules implement what is effectively a multitasking execution scheme using overlapped interrupts, accompanied by manual 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 (i.e., faster sample rate) code. Once complete, control is returned to the preempted ISR.

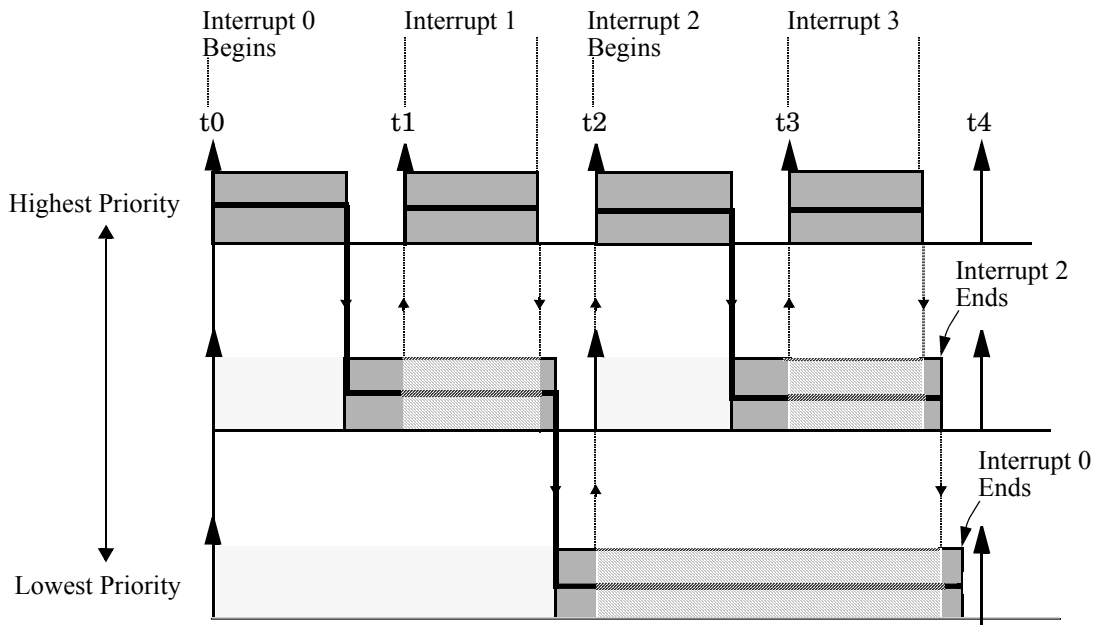
The following diagrams illustrate how mixed-rate systems are handled by Real-Time Workshop in these two environments.



-  Vertical arrows indicate sample times.
-  Dotted lines with downward pointing arrows indicate the release of control to a lower priority task.
-  Dark gray areas indicate task execution.
-  Hashed areas indicate task preemption by a higher priority task.
-  Dotted lines with upward pointing arrows indicate preemption by a higher priority task.
-  Light gray areas indicate task execution is pending.

Multitasking System Execution

The following figure illustrates how overlapped interrupts are used to implement pseudomultitasking. Note that in this case, Interrupt 0 does not return until after Interrupts 1, 2, and 3.



Pseudomultitasking Using Overlapped Interrupts

Building the Program for Multitasking Execution

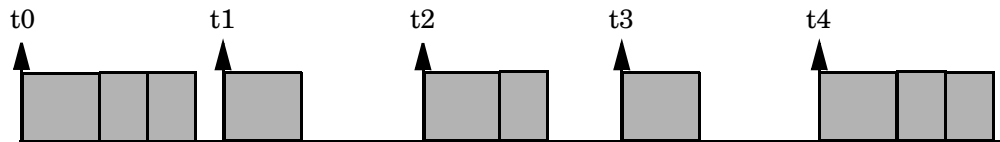
To use multitasking execution, select `Auto` (the default) or `MultiTasking` as the **Tasking mode for periodic sample times** on the **Solver** pane of the **Configuration parameters** dialog box. The **Tasking mode for periodic sample times** menu is only active if you have selected `Fixed-step` as the Solver options type. `Auto` solver mode results in a multitasking environment if your model has two or more different sample times. Note that a model with a continuous and a discrete sample time will run in single-tasking mode if the fixed-step size is equal to the discrete sample time.

Single-tasking

It is possible to execute the model code in a strictly single-tasking manner. While this method is less efficient with regard to execution speed, in certain situations it can allow you to simplify your model.

In a single-tasking environment, the base sample rate must define a time interval that is long enough to allow the execution of all blocks within that interval.

The following diagram illustrates the inefficiency inherent in single-tasking execution.



Single-tasking System Execution

Single-tasking system execution requires a sample interval that is long enough to execute one step through the entire model.

Building the Program for Single-tasking Execution

To use single-tasking execution, select the **Single-tasking** option on the **Tasking mode for periodic sample times** pulldown on the **Solver** pane of the **Configuration Parameters** dialog box. If the solver mode is **Auto**, single-tasking 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

Model Execution

To generate code that executes correctly in real time, you (or Simulink) might need to identify and properly handle sample rate transitions within the model. In multitasking mode, Simulink flags errors during simulation if the model contains illegal rate transitions. To avoid raising such errors, insert **Rate Transition** blocks between tasks. You can instruct Simulink to handle rate transitions automatically between periodic tasks by inserting hidden **Rate Transition** blocks. See “Automatic Rate Transition” on page 8-16 for an explanation of this option.

To understand such problems, first consider how Simulink simulations differ from real-time programs.

Simulating Models with Simulink

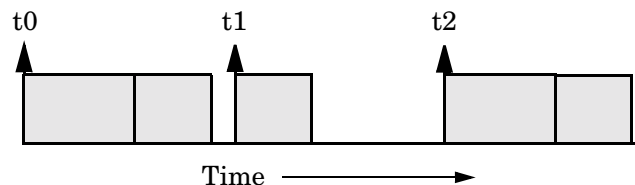
Before Simulink simulates a model, it orders all the blocks based upon their topological dependencies. This includes expanding subsystems into the individual blocks they contain and flattening the entire model into a single list. Once this step is complete, each block is executed in order.

The key to this process is the proper ordering of blocks. Any block whose output is directly dependent on its input (i.e., any block with direct feedthrough) cannot execute until the block driving its input has executed.

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, all necessary computations are performed prior to advancing the variable corresponding to time. In essence, this results in all computations occurring instantaneously (i.e., no computational delay).

Executing Models 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 Simulink), which leads to less efficient execution.



Unused Time in Sample Interval

Sample interval t_1 cannot be compressed to increase execution speed because by definition, sample times are clocked in real time.

Real-Time Workshop application programs are designed to circumvent this potential inefficiency by using a multitasking scheme. This technique defines tasks with different priorities to execute parts of the model code that have different sample rates.

See “Multitasking and Pseudomultitasking” on page 8-6 for a description of how this works. It is important to understand that section before proceeding here.

Single-tasking vs. 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.

The use of multitasking 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.

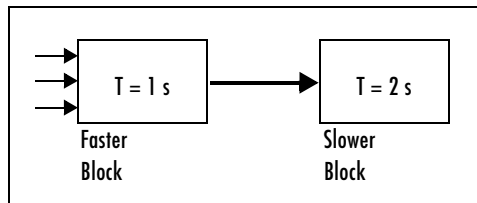
If you have a model that can benefit from multitasking execution, you might need to modify your Simulink model by adding Rate Transition blocks (or instruct Simulink to do so) to generate correct results. The next section, “Sample Rate Transitions” on page 8-12, discusses issues related to rate transition blocks.

Sample Rate Transitions

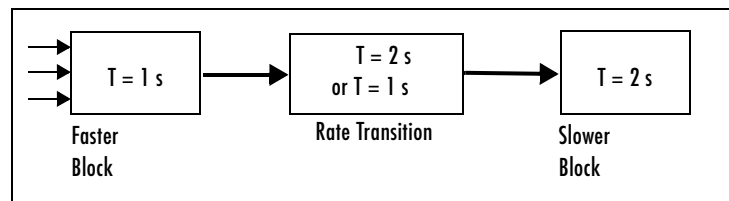
There are two possible sample rate transitions that can exist within a model:

- A faster block driving a slower block
- A slower block driving a faster block

In single-tasking systems, there are no issues involving multiple sample rates. In multitasking and pseudomultitasking systems, however, differing sample rates can cause problems by causing blocks to be executed in the wrong order. To prevent possible errors in calculated blocks data, you must control model execution at these transitions. When connecting faster and slower blocks, you or Simulink must add Rate Transition blocks between them. For faster-to-slower transitions, this diagram



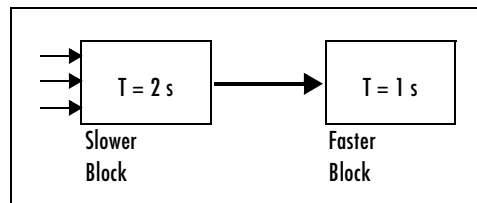
becomes



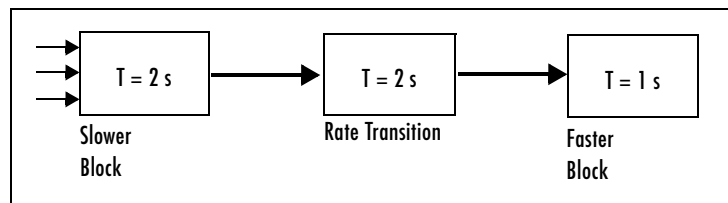
Rate Transition from Faster to Slower Blocks (T = sample period)

The sample time of fast-to-slow Rate Transition blocks can depend on the block's mode (port-based or mixed sample times).

When transitioning from slower to faster blocks, this diagram



becomes



Rate Transition from Slower to Faster Blocks ($T = \text{Sample Period}$)

The sample time of slow-to-fast Rate Transition blocks can depend on the block's mode (port-based or mixed sample times).

Data Transfer Problems

Rate Transition blocks are designed to deal with the following problems that occur in 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. These 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*. The figure “Time Overlaps in Faster to Slower Transitions (T = Sample Time)” on page 8-18 illustrates an unprotected data transfer.

In a *protected* data transfer, the output V_1 of the faster block is held until the slower block finished executing.

- *Deterministic vs. 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.

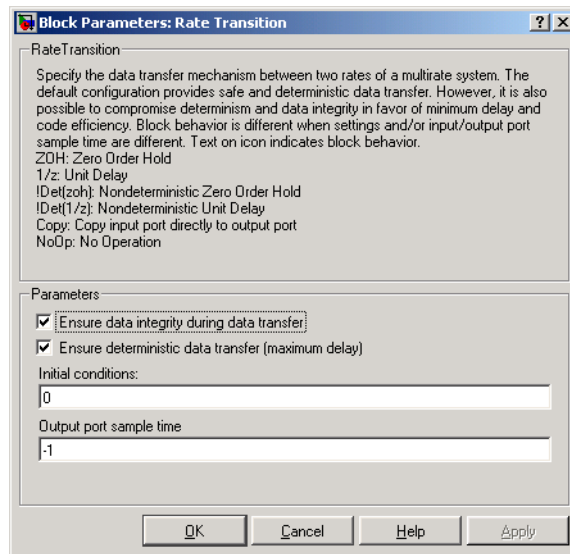
You can use the Rate Transition block to ensure that data transfers in your application are both protected and deterministic. These characteristics are considered desirable in most applications. However, the Rate Transition block supports flexible options that allow you to compromise data integrity and determinism in favor of lower latency. The next section summarizes these options.

Rate Transition Block Options

Several parameters of the Rate Transition block are relevant to its use in code generation for real-time execution. These are discussed below. For full documentation of the Rate Transition block and its block parameters, see the “Simulink Blocks” section of Using Simulink.

The Rate Transition block handles both types of transitions (fast to slow and slow to fast). When inserted between two blocks of differing sample rates, the Rate Transition block automatically configures its input and output sample rates for the appropriate type of transition; you need not specify whether a transition is slow-to-fast or fast-to-slow (low-to-high or high-to-low task priorities).

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 dialog shown below indicates, the data transfer mechanism is controlled by two options:



- **Ensure data integrity during data transfer:** When this option is on, the integrity of data transferred between rates is guaranteed (the data transfer is protected). When this option is off, data integrity is not guaranteed (the data transfer is unprotected). By default, **Ensure data integrity during data transfer** is on.
- **Ensure deterministic data transfer (maximum delay):** This option is enabled only for protected data transfer (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.

Thus the Rate Transition block offers three modes of operation with respect to data transfer. In order of safety, from safest to least safe, these are

- **Protected/Deterministic (default):** This is the safest mode. The drawback of this mode is that it introduces latency into the system:

- Fast to slow transition: maximum latency is one sample period of the slower task.
- Slow to fast transition: maximum latency is two sample periods of the slower task.
- **Protected/NonDeterministic:** In this mode, for slow-to-fast transitions, data integrity is protected by double-buffering data transferred between rates. For fast-to-slow transitions, a semaphore flag is used. The blocks downstream from the Rate Transition block always 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 the least safe, and 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 needed. 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, does generate smaller 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.

Automatic Rate Transition

Simulink can detect rate transitions in a model and automatically insert Rate Transition blocks to handle them. To instruct Simulink to do this, select **Automatically handle data transfers between tasks** on the **Solver** pane of the **Configuration parameters** dialog.

Note This option is only available for fixed-step solvers in multitasking mode.

The **Automatically handle data transfers between tasks** option is off by default. When you select it,

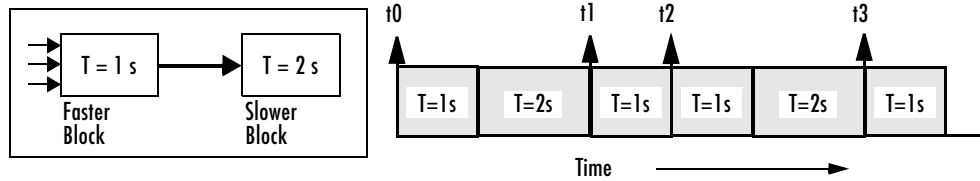
- Simulink handles all transitions between periodic sample times; no transitions between asynchronous tasks are handled.
- Simulink inserts “hidden” Rate Transition blocks that are not visible on the block diagram.
- Real-Time Workshop generates code for autoinserted Rate Transition blocks which is identical to that generated for manually-inserted Rate Transition blocks.
- Autoinserted Rate Transition blocks operate in protected/deterministic mode (see above section), which cannot be altered; if you want other modes, you should manually insert Rate Transition blocks and set their modes yourself. The optional HTML code generation report includes a list of Rate Transition blocks that Simulink auto-inserted in your model.

The next four sections describe cases in which Rate Transition blocks are necessary for 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 Simulink

In a model where a faster block drives a slower block having direct feedthrough, the outputs of the faster block are always computed first. In simulation intervals where the slower block does not execute, the simulation progresses more rapidly because there are fewer blocks to execute.

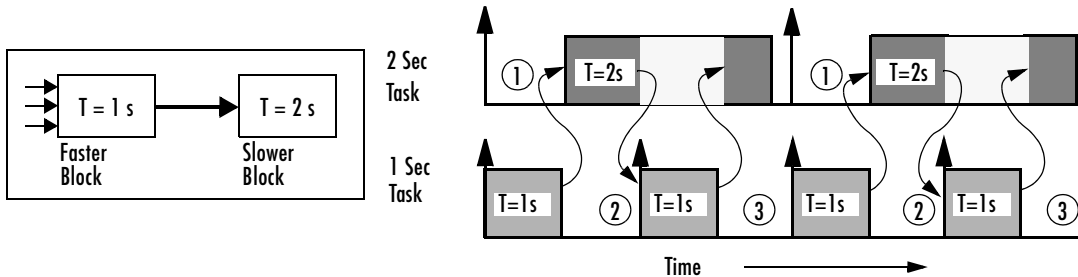
The following diagram illustrates this situation.



Simulink does not execute in real time, which means that it is not bound by real-time constraints. Simulink waits for, or moves ahead to, whatever tasks are necessary 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 following diagram illustrates a situation where this problem arises. The hashed area indicates times when tasks are preempted by higher priority before completion.

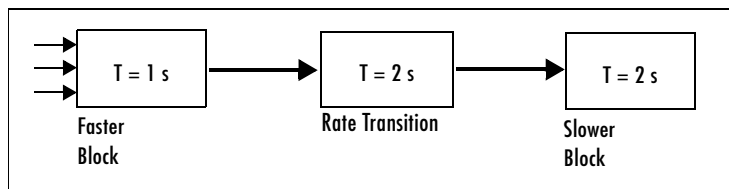


- ① The faster task ($T=1s$) completes.
- ② Higher priority preemption occurs.
- ③ The slower task ($T=2s$) resumes and its inputs have changed. This leads to unpredictable results.

Time Overlaps in Faster to Slower Transitions ($T = \text{Sample Time}$)

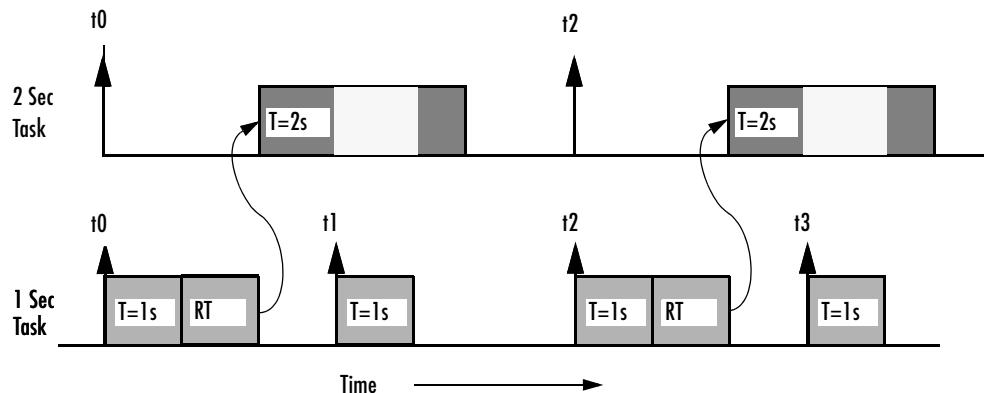
In the above figure, the faster block executes a second time before the slower block has completed execution. This can cause unpredictable results because the input data to the slow task is changing. Data integrity is not guaranteed in this situation.

To avoid this situation, Simulink 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. This guarantees that the input to the slower block does not change during its execution, ensuring 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.



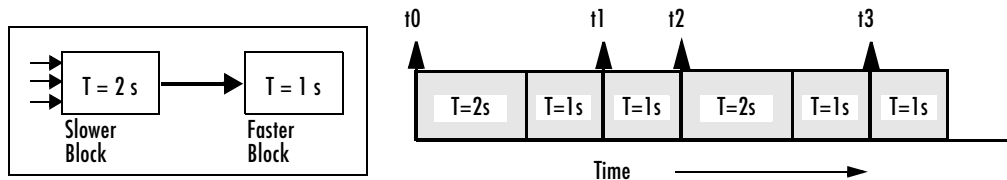
Rate Transition Blocks Preventing Fast-to-Slow Timing Overlaps

Adding a Rate Transition block ensures that the Rate Transition block executes before the 2 second block (its priority is higher) and that its output value is held constant while the 2 second block executes (it executes at the slower sample rate).

Slower to Faster Transitions in Simulink

In a model where a slower block drives a faster block, Simulink again computes the output of the driving block first. During sample intervals where only the faster block executes, the simulation progresses more rapidly.

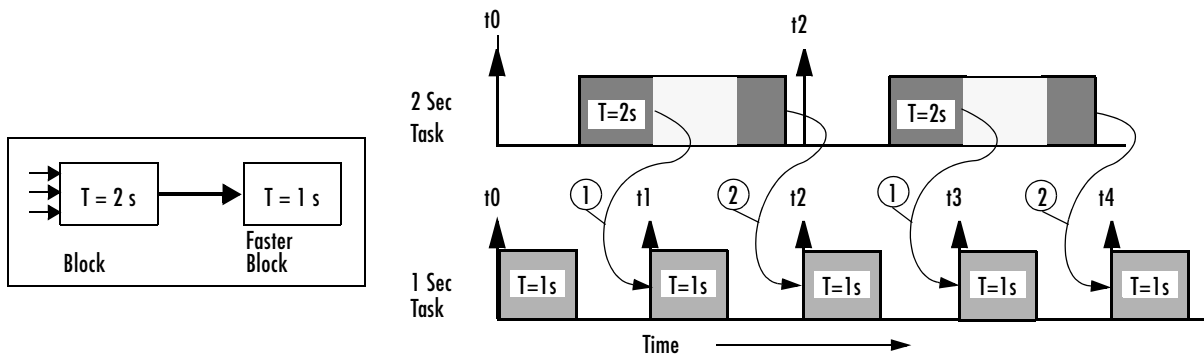
The following diagram illustrates the execution sequence.



As you can see from the preceding diagrams, Simulink can simulate models with multiple sample rates in an efficient manner. However, Simulink 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.



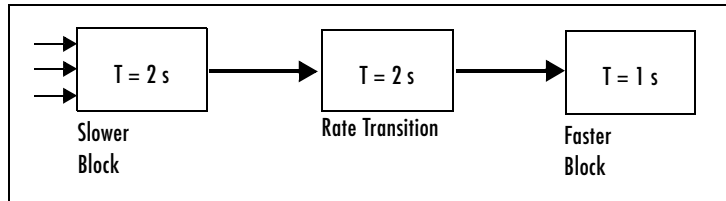
- ① The faster block executes a second time prior to the completion of the slower block.
- ② The faster block executes before the slower block.

Time Overlaps in Slower to Faster Transitions

This timing diagram illustrates two problems:

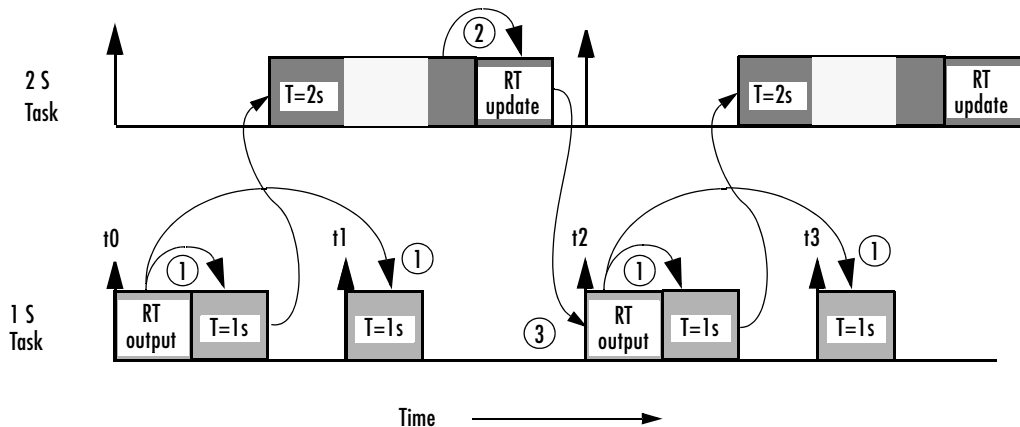
- Execution of the slower block is split over more than one faster block interval. In this case the faster task executes a second time before the slower task has completed execution. This means the inputs to the faster task can have incorrect values some of the time.
- The faster block executes before the slower block (which is backward from the way Simulink 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 picture below shows the timing sequence that results with the added Rate Transition block



Rate Transition Blocks Preventing Slow-to-Fast Timing Overlaps

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.

Single-tasking and Multitasking Execution of a Model: an Example

This section examines how a simple multirate model executes in both real time and simulation, using a fixed-step solver. It considers the operation of both Single-tasking and MultiTasking solver modes.

The example model is shown in the following figure. The discussion refers to the six blocks of the model as A through F, as labeled in the block diagram.

Note that 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. (See “Determining Block Update Order” in the Using Simulink documentation.)

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.

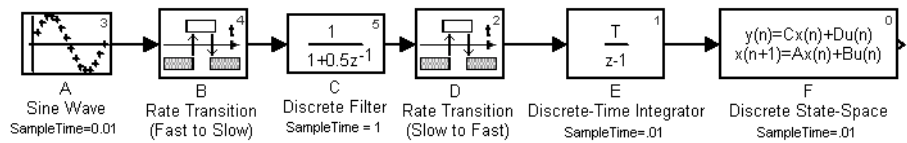


Figure 8-1: Example Model with Multiple Rates and Transition Blocks

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.

Single-tasking Execution

This section considers the execution of the model when the solver mode is `SingleTasking`.

Note that in a single-tasking system, if the **Block reduction** option is on, fast-to-slow Rate Transition blocks are optimized out of the model. We show the default case (**Block reduction** on); therefore block B does not appear in the timing diagrams in this section.

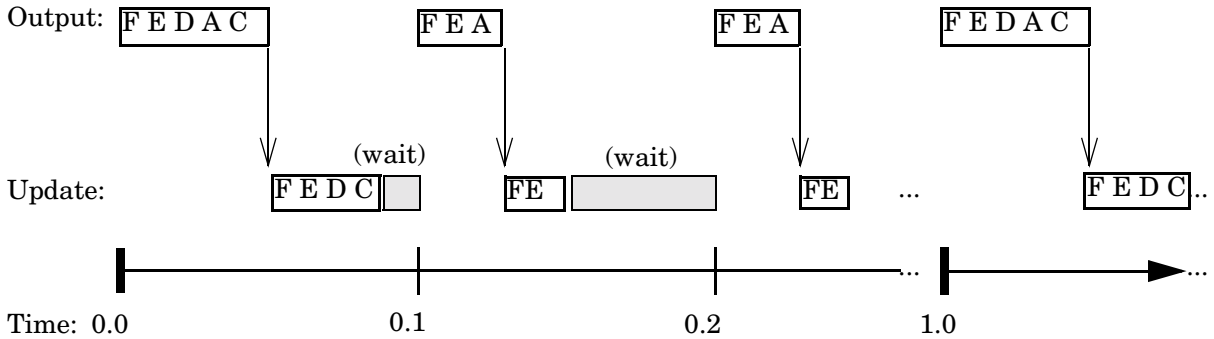
The following table shows, for each block in the model, the execution order, sample time, and whether the block has an output or update computation. Block A does not have discrete states, and accordingly does not have an update computation.

Execution Order and Sample Times (Single-tasking)

Blocks (in Execution Order)	Sample Time (in seconds)	Output	Update
F	0.1	Y	Y
E	0.1	Y	Y
D	1	Y	Y
A	0.1	Y	N
C	1	Y	Y

Real-Time Single-tasking Execution

The following 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.



Single-tasking Execution of Model in a Real-Time System

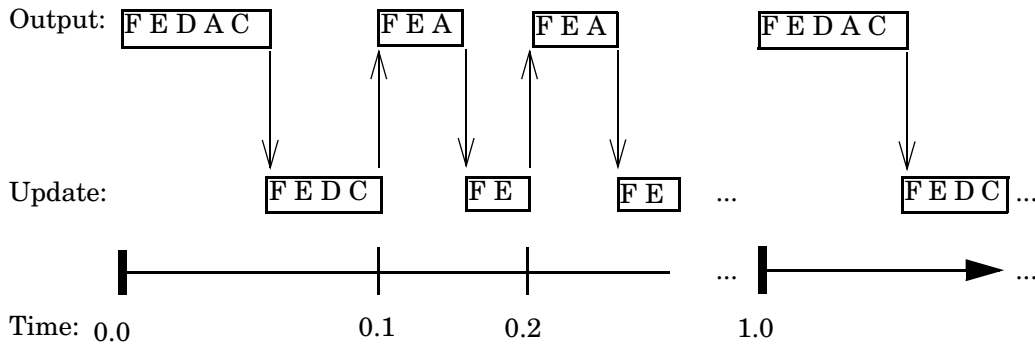
At time 0.0, 1.0, and every second thereafter, both the slow and fast blocks execute their output computations; this is followed by update computations for blocks that have states. Within a given time interval, output and update computations are sequenced in block execution order.

The fast blocks execute on every tick, at intervals of 0.1 second. Output computations are followed by update computations.

Note that 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 SingleTasking mode.

Simulated Single-tasking Execution

The following figure shows the execution of the model in Simulink via the simulation loop.



Single-tasking Execution of Model in Simulink

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.

Multitasking Execution

This section considers the execution of the model when the solver mode is `MultiTasking`. Block computations are executed under two tasks, prioritized by rate:

- The slower task, which gets 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.

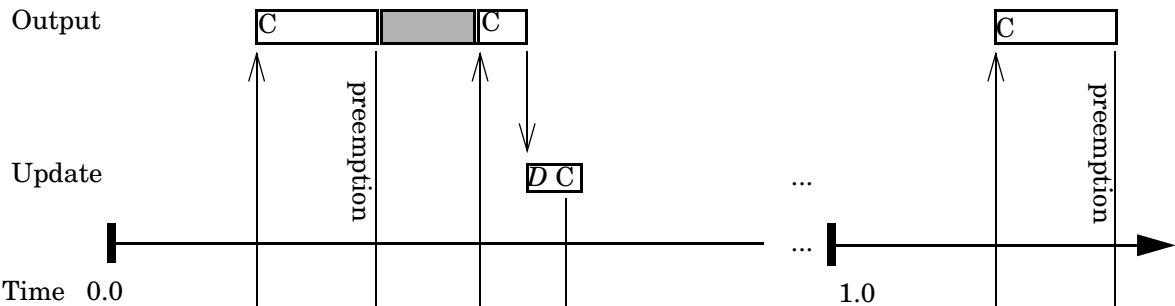
Task Allocation of Blocks in Multitasking Execution

Blocks (in Execution Order)	Task	Output	Update
F	0.1 second task	Y	Y
E	0.1 second task	Y	Y
<i>D</i>	<i>Output promoted to run under 0.1 second task (see “Block Priority Promotions”) Update runs under 1 second task</i>	Y	Y
A	0.1 second task	Y	N
<i>B</i>	<i>Promoted to run under 0.1 second task (see “Block Priority Promotions”)</i>	Y	N
C	1 second task	Y	Y

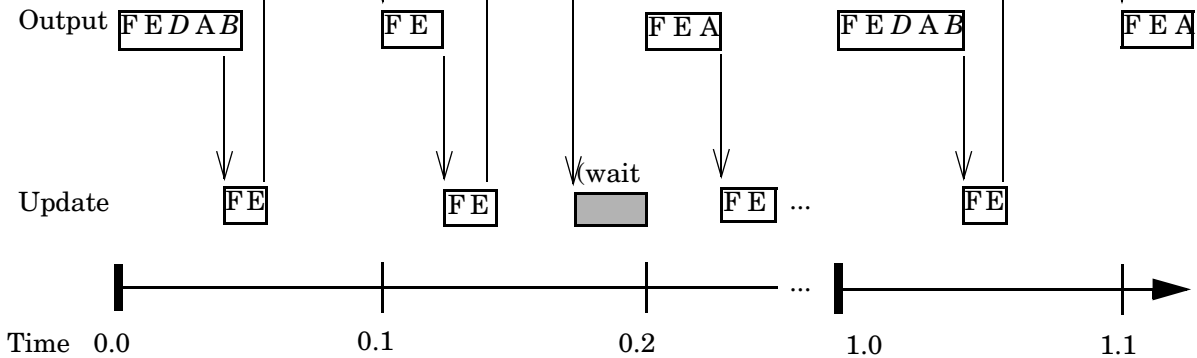
Real-Time Multitasking Execution

The following 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.

1 SECOND



0.1 SECOND



Multitasking Execution of Model in a Real-Time System

Block Priority Promotions. Notice following block “promotions”:

- The Rate Transition block B has been promoted to run at higher task priority, under the 0.1 second task. However, B still executes only at 1-second intervals, (that is, at every 10th tick of the 1-second task). In other words, B runs at the higher priority but at the slower rate.
This promotion is required because C requires input from B. Running B at higher task priority ensures that the output computation of B is always completed before C needs it.
- The *output* computation for Rate Transition block D has also been promoted to run at higher task priority, under the 0.1 second task. Like B, D’s output still executes only at 1 second intervals.
- The *update* computation for block D runs under the lower priority 1 second task, at the same priority as C. This is because the state of D is dependent upon the output of C.

On each tick, all the outputs and updates for the faster blocks must run before the lower priority block (C) gets any run time. Only block C runs entirely in the 1 second task. In the above figure, C does not complete its output computation within the first 0.1 second tick, so it is preempted by the higher priority task at time 0.1. C then resumes and completes, at which point the update function for D is executed. There is then some idle time before the next tick.

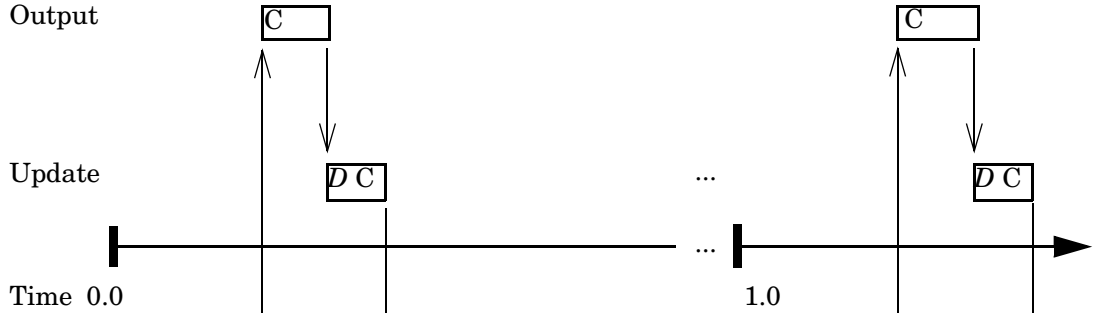
If the computations for block C were to take longer than 1 second, an interrupt overflow error condition would exist.

Notice that in multitasking mode, the program makes more efficient use of time than in single-tasking mode, as it spends less time in an idle state.

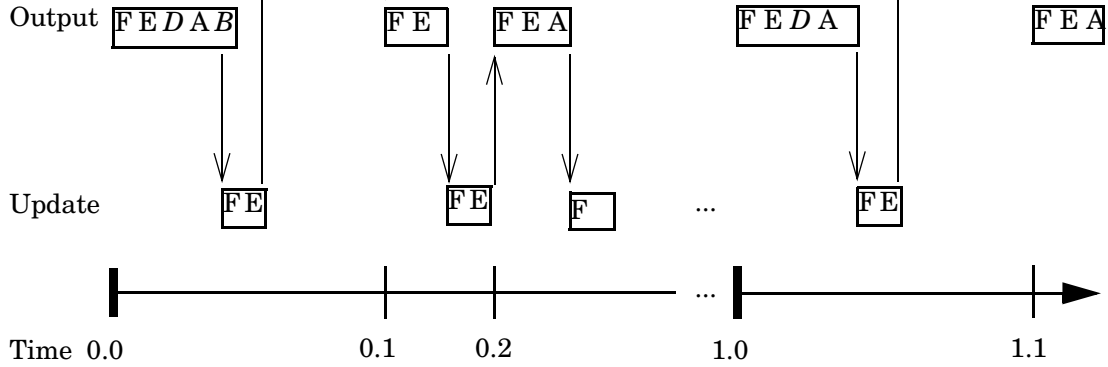
Simulated Multitasking Execution

The following figure shows the execution of the same model in Simulink, in `MultiTasking` solver mode. In this case, Simulink runs all blocks in one thread of execution, simulating multitasking. No preemption occurs.

**1 SECOND
BLOCKS**



**0.1 SECOND
BLOCKS**



Multitasking Execution of Model in Simulink

Optimizing the Model for Code Generation

You can optimize memory usage and performance of code generated from your model by Real-Time Workshop a number of ways. This chapter discusses optimization techniques that are common to all target configurations and code formats. For an overview of controlling optimization, see “Optimization Options” on page 2-17. For optimizations specific to a particular target configuration, see the chapters relevant to that target.

Optimization Parameters Overview
(p. 9-2)

Options on the **Configuration Parameters** dialog box that affect code size and efficiency

Optimizing Models (p. 9-4)

Optimization tools and techniques that you can use with any target configuration

Expression Folding (p. 9-10)

An optimization that significantly reduces the need to compute and store temporary results

Conditional Input Execution (p. 9-16)

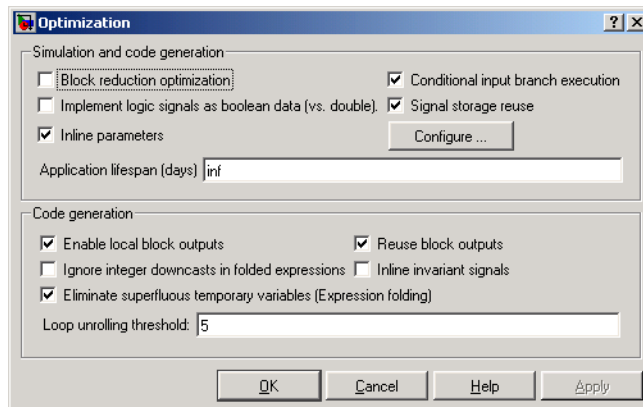
An optimization for executing inputs to switch blocks only as often as required

Block Diagram Performance Tuning
(p. 9-17)

How to use lookup tables, accumulator constructs, and data types efficiently

Optimization Parameters Overview

Many options on the **Optimization** pane of the **Configuration Parameters** dialog box affect the generated code. The dialog shown below shows the default optimization settings, plus **Inline parameters** and **Inline invariant signals** (which are both off by default).



Some basic optimization suggestions are given below, cross-referenced to more extensive relevant discussions in the documentation.

- Turn on the **Signal storage reuse** option. This directs Real-Time Workshop to store signals in reusable memory locations. It also enables the **Local block outputs** option (see below, and “Signal Storage Reuse” on page 2-22).
Disabling **Signal storage reuse** makes all block outputs global and unique, which in many cases significantly increases RAM and ROM usage.
- Select the **Inline parameters** check box. Inlining parameters reduces global RAM usage, because parameters are not declared in the global parameters structure. You can override the inlining of individual parameters by using the **Model Parameter Configuration** dialog box. You tune parameters used in referenced models differently, by declaring them as Model block parameter arguments, rather than using the **Model Parameter Configuration** dialog box. See “Inline Parameters” on page 2-20 and “Using Model Arguments” in the Simulink documentation for more details.

- Set an appropriate **Loop unrolling threshold**. The loop unrolling threshold determines when a wide signal should be wrapped into a for loop and when it should be generated as a separate statement for each element of the signal. See “Loop Unrolling Threshold” on page 2-24 for details on this feature.
- Select the **Inline invariant signals** option. Real-Time Workshop does not generate code for blocks with a constant (invariant) sample time. You must select **Inline parameters** in order to invoke this option. See “Inline Invariant Signals” on page 2-23.
- Select the **Enable local block outputs** option. Block signals are declared locally in functions instead of being declared globally (when possible). You must select **Signal storage reuse** to enable the **Enable local block outputs** option. See “Enable Local Block Outputs” on page 2-23.
- Select the **Eliminate superfluous temporary variables (Expression folding)** option, discussed in “Expression Folding” on page 9-10.
- Select the **Reuse block outputs** option. This option can reduce stack size where signals are being buffered in local variables. See “Reuse Block Outputs” on page 2-23.

Optimizing Models

Using the Model Advisor

The Model Advisor comprehensively analyzes your model to help you appropriately configure Simulink and Real-Time Workshop. This tool analyzes model and block-level configuration and reports findings to the screen. You can enable or disable any specific check. Model Advisor is particularly useful early in the design cycle.

Using the Model Advisor, you can quickly analyze a model for code generation and identify aspects of your model that impede production deployment or limit code efficiency. You control the tool from a browser window, from which you can select from a set of checks on a model's current configuration. Model Advisor analyzes the model and in the same window generates a report describing both good and bad conditions, 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.

Running the Model Advisor

The Model Advisor can be launched in several ways:

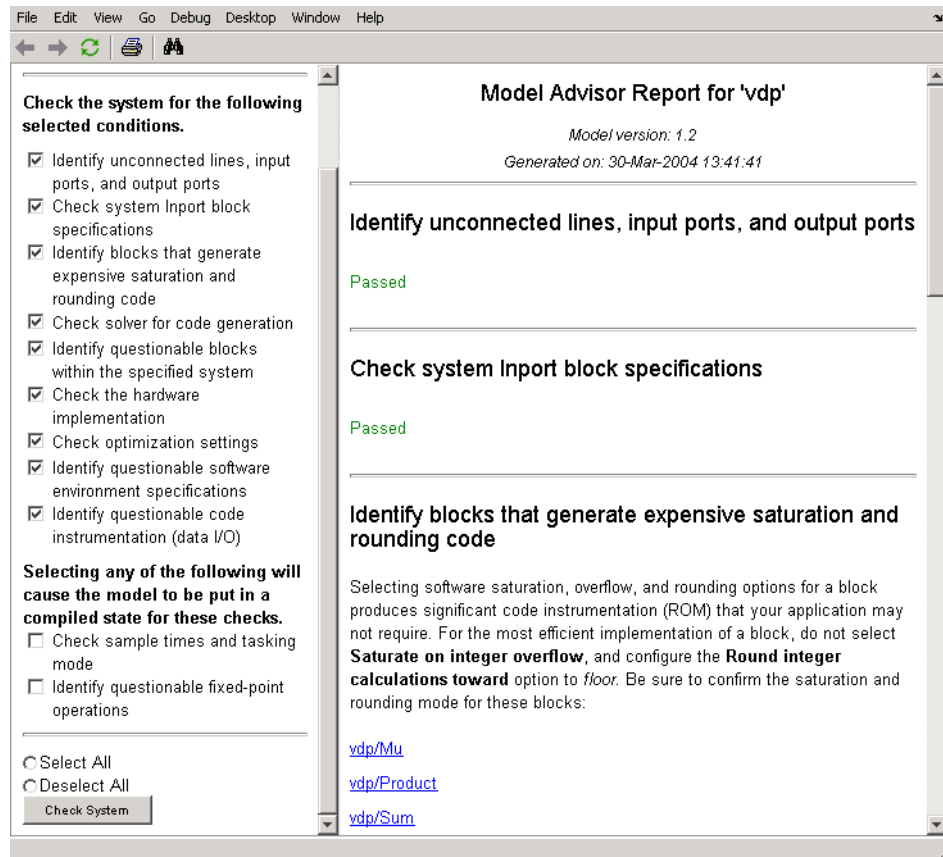
- From a Simulink model, choose **Model Advisor** from the **Tools** menu to start it for that model.
- From Model Explorer, select the node `Advice` for `model` in the **Contents** pane to start Model Advisor for that model.
- From the MATLAB prompt, type
 - `modeladvisor('model')` invokes the tool for the root system of a model. If the specified model is not currently open, Model Advisor opens it.
 - `modeladvisor('model/system')` invokes the tool for the specified system. If the specified model is not currently open, Model Advisor opens it.

You can also invoke the Model Advisor for models and systems using the built-in Simulink `bdroot`, `gcb`, and `gcs` handles. For example,

```
modeladvisor(gcb)
```

invokes the Model Advisor for the currently selected block (which would normally be a subsystem).

The Model Advisor window is shown below, containing a report for the vdp demo model. The left scrolling pane lists the checks you can select individually or completely; After you click **Check System**, Model Advisor places the results of the checks in the right scrolling pane:



Persistence of Reports. Reports generated from the Model Advisor are persistent (they are saved inside the slprj project directory under the modeladvisor directory). Thus, if the Model Advisor is opened for a model that has been previously checked, the previous report is visible (as long as you have not changed your working directory). Checking the model again with the Model Advisor replaces that report.

Recommended Usage

The Model Advisor complements both Simulink and Real-Time Workshop, and works most effectively with ERT and ERT-based targets (targets based on Real-Time Workshop Embedded Coder), although it will operate with any target. Select the target you plan to use in the **Real-Time Workshop** pane of the **Configuration Parameters** dialog before using the Model Advisor.

Model Advisor Demo Models

You can run three demo models for Model Advisor:

- `rtwdemo_advisor1`
- `rtwdemo_advisor2`
- `rtwdemo_advisor3`

You can also access these demos by opening the general Real-Time Workshop demo package (`rtwdemos`) and navigating to the Model Advisor-related models in the “Optimizations” group. You can also type, for example,

```
modeladvisor('rtwdemo_advisor1')
```

to launch the `rtwdemo_advisor1` model. These demos are intended to show the various aspects of model and block configuration. Note that demo models `rtwdemo_advisor2` and `rtwdemo_advisor3` require Stateflow and Fixed-Point Toolbox.

Demos Illustrating Optimizations

The `rtwdemos` demo suite includes a set of demonstration models that illustrate optimization settings and techniques. To access these demos, type

```
rtwdemos
```

or click the above command. When the top-level model titled “Real-Time Workshop and Real-Time Workshop Embedded Coder Feature Demonstrations” opens, double-click the **Optimization** box to open the Optimizations demo:

Optimization

open	Inlining block parameters, and propagating constant values
open	Expression folding in Simulink
open	Expression folding in Stateflow
open	Conditional branch execution
open	Dead-path elimination
open	For-loop fusion
open	Timer/counters for absolute and elapsed time
open	Temporary variables for merged signals
open	Model Advisor for optimizing the diagram for real-time deployment

Real-Time Workshop and Real-Time Workshop Embedded Coder perform countless optimizations, of which these examples are a tiny subset. Real-Time Workshop Embedded Coder, designed for production code deployment, further optimizes the generated code where RAM, ROM and CPU constraints are paramount.

Use these demos to learn about the specific effects that optimization parameters and techniques have on models. You can also access the Model Advisor demos from the **Optimization** demo page.

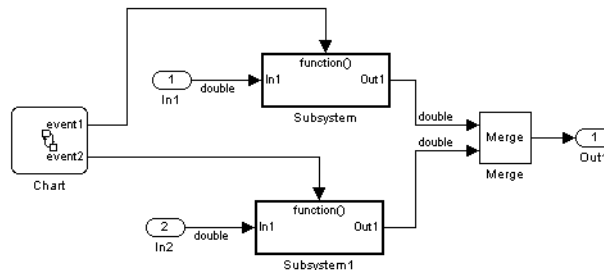
Other Optimization Tools and Techniques

In addition to analyzing models with Model Advisor (see “Using the Model Advisor” on page 9-4), you can use a variety of other tools and techniques that work with any code format. Here are some particularly useful ones:

- Run the `slupdate` command to automatically convert older models (saved by prior versions or by the current one) to use current features. For details about what `slupdate` does, type

```
help slupdate
```
- Before building, set optimization flags for the compiler (e.g., `-O2` for gcc, `-O1` for Microsoft Visual C).

- Directly inline C code S-functions into the generated code by writing a TLC file for the S-function. See Chapter 10, “Writing S-Functions for Real-Time Workshop” and the Target Language Compiler documentation for more information on inlining S-functions.
- 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). See Using Simulink for more information on data types. For a block-by-block summary, you can type `showblockdatatypetable` in MATLAB, or click the command above.
- 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 Stateflow. The model below shows an example of how to use the Merge block.



Minimizing Memory Requirements for Parameters and Data During Code Generation

When Real-Time Workshop generates code, it creates an intermediate 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, (e.g., 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 performance 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, Real-Time Workshop places a special reference key in the intermediate file that enables the Target Language Compiler to access the data directly from Simulink when it is needed 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 Real-Time Workshop to represent it literally in the *model.rtw* file. Whenever this threshold size is exceeded, Real-Time Workshop 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.

Expression Folding

Expression folding is a code optimization technique that minimizes the computation of intermediate results at block outputs and the storage of such results in temporary buffers or variables. When expression folding is on, Real-Time Workshop collapses, or “folds,” block computations into single expressions, instead of generating separate code statements and storage declarations for each block in the model.

Expression folding can dramatically improve 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.

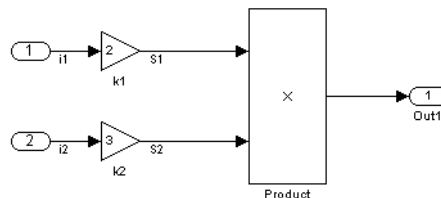
By default, expression folding is on. The Real-Time Workshop code generation options are configured to use expression folding wherever possible. Most Simulink blocks support expression folding.

You can also take advantage of expression folding in your own inlined S-function blocks. See “Supporting Expression Folding in S-Functions” on page 10-46 for information on how to do this.

In the code generation examples that follow, note that the **Signal storage reuse** optimizations (**Enable local block outputs**, **Reuse block outputs**, and **Eliminate superfluous temporary variables (Expression folding)**) are all turned on.

Expression Folding Example

As a simple example of how expression folding affects the code generated from a model, consider the model shown below.



Expression Folding Example Model

With expression folding on, this model generates a single-line output computation, as shown in this *model_output* function.

```
static void exprfld_output(int_T tid)
{
    /* local block i/o variables */

    /* Output: '<Root>/Out1' incorporates:
     * Gain: '<Root>/k1'
     * Gain: '<Root>/k2'
     * Product: '<Root>/Product'
     */
    exprfld_Y.Out1 = exprfld_U.i1 * exprfld_P.k1_Gain *
        (exprfld_U.i2 * exprfld_P.k2_Gain);
}
```

The generated comments indicate the block computations that were combined into a single expression. The comments also document the block parameters that appear in the expression.

With expression folding off, the same model computes temporary results for both Gain blocks and the Product block before the final output, as shown in this output function:

```
static void exprfld_output(int_T tid)
{
    /* local block i/o variables */

    real_T rtb_k2_i;
    real_T rtb_Product_i;

    /* Gain: '<Root>/k1' */
    rtb_Product_i = exprfld_U.i1 * exprfld_P.k1_Gain;

    /* Gain: '<Root>/k2' */
    rtb_k2_i = exprfld_U.i2 * exprfld_P.k2_Gain;

    /* Product: '<Root>/Product' */
    rtb_Product_i *= rtb_k2_i;
```

```
/* Outport: '<Root>/Out1' */  
exprfld_Y.Out1 = rtb_Product_i;  
}
```

For an example of expression folding in the context of a more complex model, [link to the exprfolding demo](#), or type the following command at the MATLAB prompt.

```
rtwdemo_slexprfold
```

Using and Configuring Expression Folding

The options described in this section let you control the operation of expression folding.

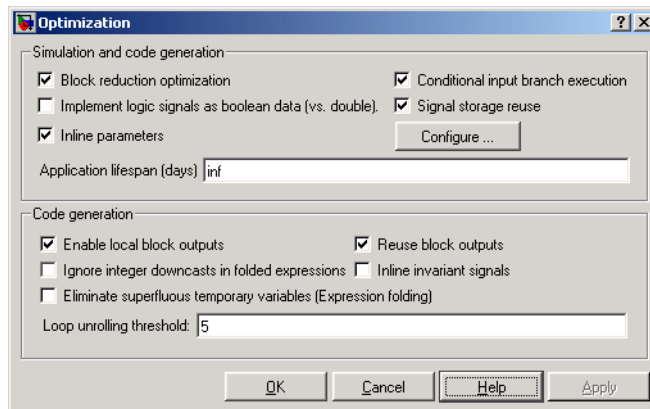
Enabling Expression Folding

Expression folding operates only on expressions involving local variables. Expression folding is therefore available only when the **Signal storage reuse** code generation option is on.

For a new model, default code generation options are set to use expression folding. If you are configuring an existing model, you can ensure that expression folding is turned on as follows:

- 1 Open the **Configuration Parameters** dialog box and select the **Optimization** pane.
- 2 Select the **Signal storage reuse** option.
- 3 Select the **Enable Local block outputs** option.
- 4 Enable expression folding by selecting **Eliminate superfluous temporary variables (Expression folding)**.

The **Optimization** pane is shown below. By default, all expression folding related options are selected, as shown.



5 Click **Apply**.

Expression Folding Options

This section discusses the optimization code generation options related to expression folding.

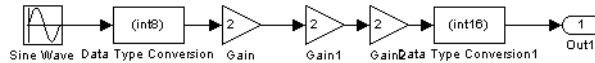
Expression Folding. This option turns the expression folding feature on or off. When **Expression folding** is selected, the **Ignore integer downcasts in folded expressions** option becomes available.

Ignore integer downcasts in folded expressions. This option specifies how Real-Time Workshop should handle 8-bit operations on 16-bit microprocessors and 8- and 16-bit operations on 32-bit microprocessors. It is available only when expression folding is being performed. To ensure consistency between simulation and code generation, the results of 8 and 16-bit integer expressions must be explicitly downcast.

Selecting this option improves code efficiency by avoiding casts of intermediate variables. However, the primary effect of selecting this option is that expressions involving 8- and 16-bit arithmetic are less likely to overflow in code than they are in simulation. Therefore, it is good practice to turn off **Ignore integer downcasts in folded expressions** for safety, to ensure that answers obtained from generated code are the same as those from simulations. Turn the option on only if you are concerned with generating the smallest possible code,

and either you know that 8- and 16-bit signals will not overflow, or you do not care if they do.

As an example, consider this model.



The following code fragment shows the output computation (within the output function) when **Ignore integer downcasts in folded expressions** is off. The Gain blocks are folded into a single expression. In addition to the typecasts generated by the Type Conversion blocks, each Gain block output is cast to `int8_T`.

```
int8_T rtb_Data_Type_Conversion;
.
.
.
rtY.Out1 = (int16_T)(int8_T)(rtP.Gain2_Gain * (int8_T)(rtP.Gain1_Gain *
(int8_T)(rtP.Gain_Gain * rtb_Data_Type_Conversion)));
```

If **Ignore integer downcasts in folded expressions** is on, the code contains only the typecasts generated by the Type Conversion blocks, as shown in the following code fragment.

```
int8_T rtb_Data_Type_Conversion;
.
.
.
rtY.Out1 = (int16_T)(rtP.Gain2_Gain * (rtP.Gain1_Gain * (rtP.Gain_Gain *
rtb_Data_Type_Conversion)));
```

Discussion. Suppose you create a model in which the output of a Sum block is a signed 8-bit number. Such numbers have a range of from -128 to +127. During simulation, the value of the Sum block's output will always be in the range -128 to +127. If the calculations involved in computing the output exceeded that range, then an overflow would occur and Simulink would provide an (optional) diagnostic.

When it comes to code running on a target processor, integer downcasts occur frequently. Most microprocessors are designed to do direct math on integers of

certain sizes. For example, a typical 16-bit microprocessor might only provide for direct multiplication on 16-bit integers and direct addition for 16- and 32-bit integers. Such a processor can perform mathoperations on smaller integers, but only indirectly, according to the following steps:

- 1 The smaller integers are loaded into bigger CPU registers.
- 2 The "big math" is performed.
- 3 The results are "integer downcast" so they are limited to the range of the smaller integers, e.g., -128 to +127

Step 3 requires extra machine instructions, ROM code, and clock cycles.

In many situations, Step 3 is a total waste of effort. For example, you might have designed your model so that it is impossible for the results to exceed the range -128 to +127. With such safeguards in place, step 3 will never change the results of calculations, and simply gives a less efficient implementation. In this type of situation you should turn on the **Ignore integer downcasts in folded expressions** optimization and bypass the range checks that decrease your application's efficiency without contributing any value.

If the calculations had overflowed, then turning (or keeping) off the **Ignore integer downcasts in folded expressions** option would cause your generated code to give different results from the Simulink model produced. You might or might not consider this difference to be a problem, but it is something you should be aware can happen.

Conditional Input Execution

Conditional input branch execution is a Simulation and code generation optimization technique that improves model execution when the model contains Switch and Multiport Switch blocks. By default, the Real-Time Workshop code generation options are configured to use the conditional input branch optimization.

When conditional input branch optimization is on, instead of executing all blocks driving the Switch block input ports at each time step, only the blocks required to compute the control input and the data input selected by the control input are executed.

Several considerations affect or limit Switch block optimization:

- Only blocks with -1 (inherited) or inf (Constant) sample time can participate in Switch block optimization.
- Blocks with outputs flagged as test points cannot participate.
- No multirate block can participate.
- Blocks with states cannot participate.
- Only S-functions with option `SS_OPTION_CAN_BE_CALLED_CONDITIONALLY` set can participate.

You control conditional input branch optimization by selecting and deselecting the **Conditional input branch execution** check box on the **Optimization** pane of the **Configuration Parameters** dialog box.

To run a conditional input branch optimization demo, use this link to the `condinputexec` demo or type the following command at the MATLAB prompt.

```
rtwdemo_condinput
```

Block Diagram Performance Tuning

Certain block constructs in Simulink will run faster, or require less code or data memory, than other seemingly equivalent constructs. Knowing the tradeoffs between similar blocks and block parameter options enables you to create Simulink models that have intuitive diagrams, and to produce the tight code that you want from Real-Time Workshop. Many of the options and constructs discussed in this section improve the simulation speed of the model itself, even without code generation.

Lookup Tables and Polynomials

Simulink provides several blocks that allow approximation of functions. These include blocks that perform direct, interpolated, and cubic spline lookup table operations, and a polynomial evaluation block.

There are currently six different blocks in Simulink that perform lookup table operations:

- Look-Up Table
- Look-Up Table (2-D)
- Look-Up Table (n-D)
- Direct Look-Up Table (n-D)
- PreLook-Up Index Search
- Interpolation (n-D) Using PreLook-Up Index Search

In addition, the Repeating Sequence block uses a lookup table operation, the output of which is a function of the real-time (or simulation-time) clock.

To get the most out of the following discussion, you should familiarize yourself with the features of these blocks, as documented in *Using Simulink*.

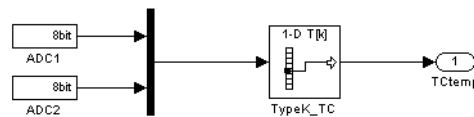
Each type of lookup table block has its own set of options and associated tradeoffs. The examples in this section show how to use lookup tables effectively. The techniques demonstrated here will help you achieve maximal performance with minimal code and data sizes.

Multichannel Nonlinear Signal Conditioning

The following figure shows a Simulink model that reads input from two 8-channel, high-speed 8-bit analog/digital converters (ADCs). The ADCs are

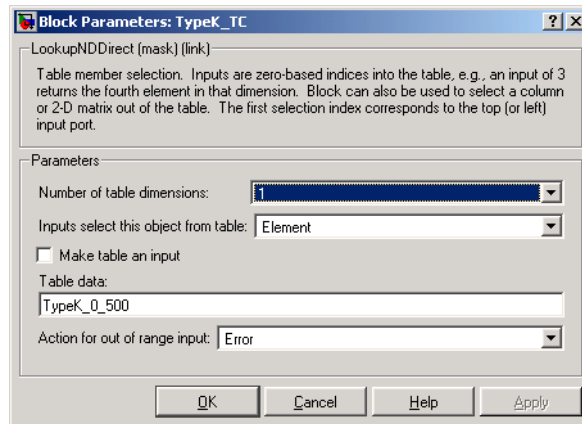
connected to Type K thermocouples through a gain circuit with an amplification of 250. Since the popular Type K thermocouples are highly nonlinear, there is an international standard for converting their voltages to temperature. In the range of 0 to 500 degrees Celsius, this conversion is a tenth-order polynomial. One way to perform the conversion from ADC readings (0-255) into temperature (in degrees Celsius) is to evaluate this polynomial. In the best case, the polynomial evaluation requires 9 multiplications and 10 additions per channel.

A polynomial evaluation is not the fastest way to convert these 8-bit ADC readings into measured temperature. Instead, the model uses a Direct Look-Up (n-D) Table block (named TypeK_TC) to map 8-bit values to temperature values. This block performs one array reference per channel.



Direct Look-Up Table (n-D) Block Conditions ADC Input

The block's table parameter has 256 values that correspond to the temperature at an ADC reading of 0, 1, 2, ... up to 255. The table data, calculated in MATLAB, is stored in the workspace variable TypeK_0_500. The block's **Table data** parameter field references TypeK_0_500, as the above figure shows.



Parameters of Direct Look-Up Table (n-D) Block

The model uses a Mux block to collect all similar signals (e.g., Type K thermocouple readings) and feed them into a single Direct Look-Up Table block. This is more efficient than using one Direct Look-Up Table block per device. If multiple blocks share a common parameter (such as the table in this example), Real-Time Workshop creates only one copy of that parameter in the generated code.

This is the recommended approach for signal conditioning when the size of the table can fit within your memory constraints. In this example, the table stores 256 double (8-byte) values, utilizing 2 KB of memory.

Note that the TypeK_TC block processes 24 channels of data sequentially.

Real-Time Workshop generates the following code for the TypeK_TC block shown previously.

```

/* LookupNDDirect: '<Root>/TypeK_TC' */
/* 1-dimensional Direct Look-Up returning 24 Scalars */
{
    int_T i1;

    const uint8_T *u0 = rtb_TmpHiddenBuffer_Feeding_Typ;
    real_T *y0 = rtb_TypeK_TC_k;

    for (i1=0; i1 < 24; i1++) {

```

```

        y0[i1] = (lookupADC_ConstP.TypeK_TC_table[u0[i1]]);
    }
}

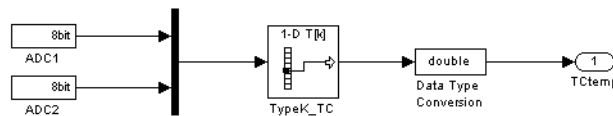
{
    int32_T i1;

    /* Outport: '<Root>/TCtemp' */
    for(i1=0; i1<24; i1++) {
        lookupADC_Y.TCtemp[i1] = rtb_TypeK_TC_k[i1];
    }
}

```

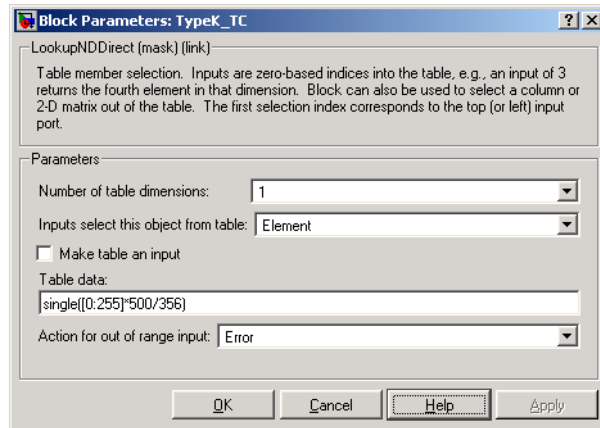
Notice that the core of each loop is one line of code that directly retrieves a table element from the table and places it in the block output variable. There are two loops in the generated code because the two simulated ADCs are not merged into a contiguous memory array in the Mux block. Instead, to avoid a copy operation, the Direct Look-Up Table block performs the lookup on two sets of data using a single table array (lookupADC_ConstP.TypeK_TC_table[]).

If the input accuracy for your application (not to be confused with the number of I/O bits) is 24 bits or less, you can use a single-precision table for signal conditioning. Then, cast the lookup table output to double precision for use in the rest of the block diagram. This technique, shown below, causes no loss of precision.



Single-Precision Lookup Table Output Is Cast to Double Precision

Note that a direct lookup table covering 24 bits of accuracy would require 64 megabytes of memory, which is typically not practical. To create a single-precision table, use the MATLAB `single()` cast function in your table calculations. Alternatively, you can perform the type cast directly in the **Table data** parameter, as shown below.



Type Casting Table Data in a Direct Lookup Block

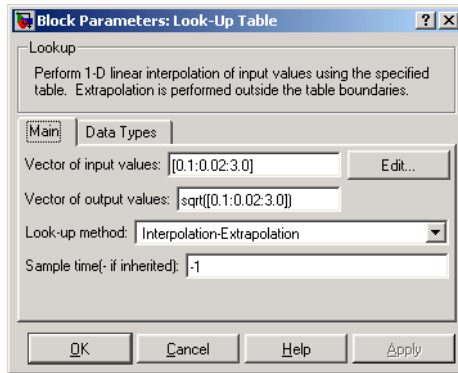
When table size becomes impractical, you must use other nonlinear techniques, such as interpolation or polynomial techniques. The Look-Up Table (n-D) block supports linear interpolation and cubic spline interpolation. The Polynomial block supports evaluation of noncomplex polynomials.

Compute-Intensive Equations

The blocks described in this section are useful for simplifying fixed, complex relationships that are normally too time consuming to compute in real time.

The only practical way to implement some compute-intensive functions or arbitrary nonlinear relationships in real time is to use some form of lookup table. On processors that do not have floating-point instructions, even functions like `sqrt()` can become too expensive to evaluate in real time.

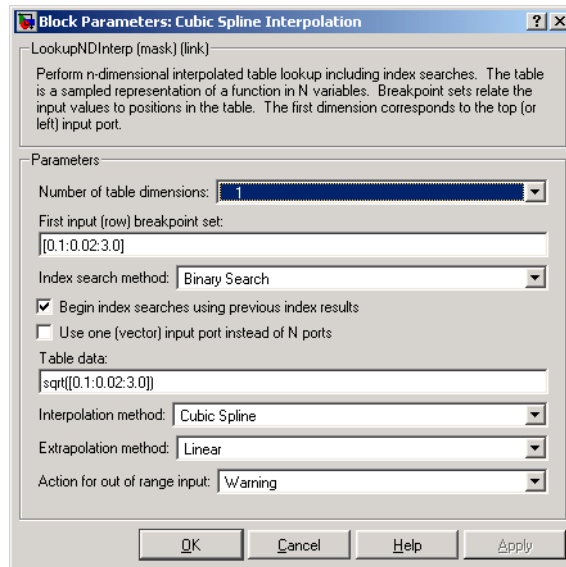
An approximation to the nonlinear relationship in a known range will work in most cases. For example, your application might require a square root calculation that your target processor's instruction set does not support. The illustration below shows how you can use a Look-Up Table block to calculate an approximation of the square root function that covers a given range of the function.



The interpolated values are plotted on the block icon.



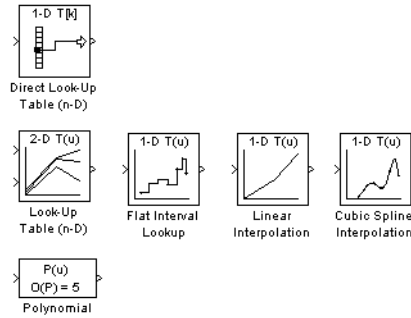
For more accuracy on widely spaced points, use a cubic spline interpolation in the Look-Up Table (n-D) block, as shown below.



Techniques available in Simulink include n-dimensional support for direct lookup, linear interpolations in a table, cubic spline interpolations in a table, and 1-D real polynomial evaluation.

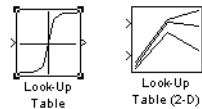
The Look-Up Table (n-D) block supports flat interval lookup, linear interpolation and cubic spline interpolation. Extrapolation for the Look-Up Table (n-D) block can either be disabled (clipping) or enabled for linear or spline extrapolations.

The icons for the Direct Look-Up Table (n-D) and Look-Up Table (n-D) blocks change depending on the type of interpolation selected and the number of dimensions in the table, as illustrated below.



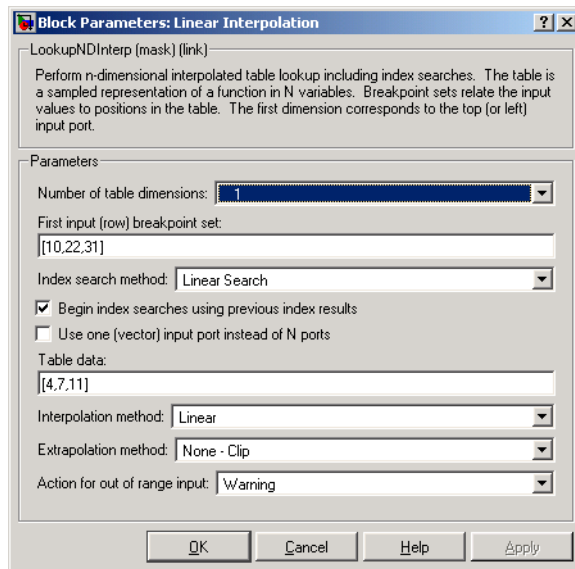
Tables with Repeated Points

The Look-Up Table and Look-Up Table (2-D) blocks, shown below, support linear interpolation with linear extrapolation. In these blocks, the row and column parameters can have repeated points, allowing pure step behavior to be mixed in with the linear interpolations. Note that this capability is not supported by the Look-Up Table (n-D) block.



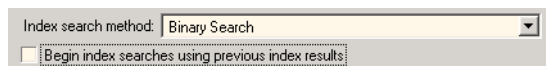
Slowly vs. Rapidly Changing Look-Up Table Block Inputs

You can optimize lookup table operations using the Look-Up Table (n-D) block for efficiency if you know the input signal's normal rate of change. The following figure shows the parameters for the Look-Up Table (n-D) block.

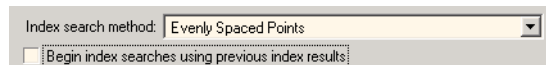


Parameter Dialog for the Look-Up Table (n-D) Block

If you do not know the input signal's normal rate of change in advance, it would be better to choose the **Binary Search** option for the index search in the Look-Up Table (n-D) block and the PreLook-Up Index Search block.



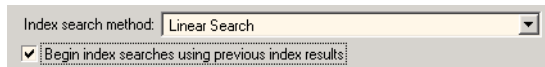
Regardless of signal behavior, if the table's breakpoints are evenly spaced, it is best to select the **Evenly Spaced Points** option from the Look-Up Table (n-D) block's parameter dialog.



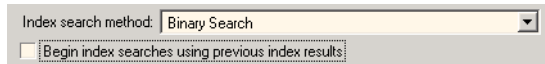
If the breakpoints are not evenly spaced, first decide which of the following best describes the input signal behavior.

- Behavior 1: The signal stays in a given breakpoint interval from one time step to the next. When the signal moves to a new interval, it tends to move to an adjacent interval.
- Behavior 2: The signal has many discontinuities. It jumps around in the table from one time step to the next, often moving three or more intervals per time step.

Given behavior 1, the best optimization for a given lookup table is to use the **Linear search option** and **Begin index searches using previous index results** options, as shown below.

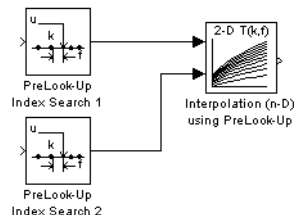


Given behavior 2, the **Begin index searches using previous index results** option does not necessarily improve performance. Choose the **Binary Search** option, as shown below.



The choice of an index search method can be more complicated for lookup table operations of two or more dimensions with linear interpolation. In this case, several signals are input to the table. Some inputs may have evenly spaced points, while others can exhibit behavior 1 or behavior 2.

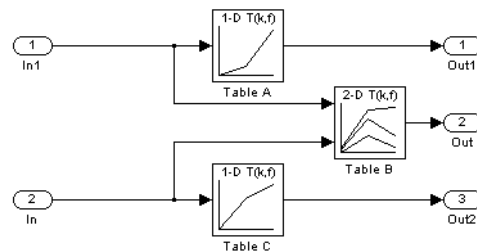
Here it might be best to use PreLook-Up Index Search blocks with different search methods (evenly spaced, linear search, or binary search) chosen according to the input signal characteristics. The outputs of these search blocks are then connected to an Interpolation (n-D) Using PreLook-Up Index Search block, as shown in the block diagram below.



You can configure each PreLook-Up Index Search block independently to use the best search algorithm for the breakpoints and input time variation cases.

Multiple Tables with Common Inputs

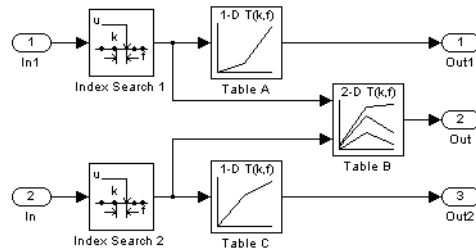
The index search can be the most time consuming part of flat or linear interpolation calculations. In large block diagrams, lookup table blocks often have the same input values as other lookup table blocks. If this is the case in your block diagram, you can save much computation time by making the breakpoints common to all tables. This saving is obtained by using one set of PreLook-Up Index Search blocks to perform the searches once for all tables, so that only the interpolation remains to be calculated. The following figure is an example of a block diagram that can be optimized by this method.



Before Optimization

Assume that Table A's breakpoints are the same as Table B's first input breakpoints, and that Table C's breakpoints are the same as Table B's second input breakpoints.

A 50% reduction in index search time is obtained by pulling these common breakpoints out into a pair of PreLook-Up Index Search blocks, and using Interpolation (n-D) Using PreLook-Up Index Search blocks to perform the interpolation. The following figure shows the optimized block diagram.



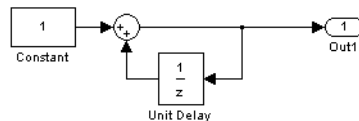
After Optimization

In the above diagram, the Look-Up Table (n-D) blocks have been replaced with Interpolation (n-D) Using PreLook-Up blocks. The PreLook-Up Index Search blocks have been added to perform the index searches separately from the interpolations, in order to realize the savings in computation time.

In large controllers and simulations, it is not uncommon for hundreds of multidimensional tables to rely on a dozen or so breakpoint sets. Using the optimization technique shown in this example, you can greatly increase the efficiency of your application.

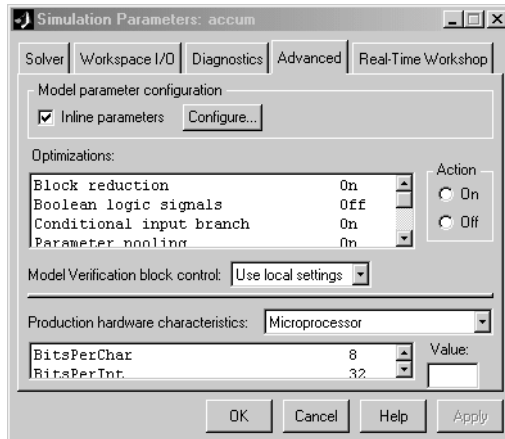
Accumulators

Simulink recognizes the block diagram shown in the following figure as an accumulator. An accumulator construct — comprising a Constant block, a Sum block, and feedback through a Unit Delay block — is recognized anywhere across a block diagram, or within subsystems at lower levels.



An Accumulator Algorithm

By using the **Block reduction** option, you can significantly optimize code generated from an accumulator. Turn this option on in the **Advanced** pane of the Simulink **Configuration Parameters** dialog, as shown below.



Block Reduction Option

With the **Block reduction** option on, Simulink creates a synthesized block, `Sum_synth_accum`. This synthesized block replaces the previous block diagram, resulting in a simple increment calculation.

```
static void accum_output(int_T tid)
{
    /* UnitDelay Block: '<Root>/Unit Delay'
     *   Operating as an accumulator
     */

    accum_DWork.UnitDelay_DSTATE++;
    accum_B.UnitDelay_j = accum_DWork.UnitDelay_DSTATE;

    /* Outputport: '<Root>/Out1' */
    accum_Y.Out1 = accum_B.UnitDelay_j;
}
```

With **Block reduction** turned off, the generated code reflects the block diagram more literally, but less efficiently.

```
static void accum_output(int_T tid)
{
    /* UnitDelay: '<Root>/Unit Delay' */
    accum_B.UnitDelay_j = accum_DWork.UnitDelay_DSTATE;

    /* Sum: '<Root>/Sum' */
    accum_B.Sum_l = 1.0 + accum_B.UnitDelay_j;

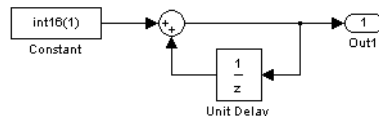
    /* Outport: '<Root>/Out1' */
    accum_Y.Out1 = accum_B.Sum_l;
}
```

Use of Data Types

In most processors, the use of integer data types can result in a significant reduction in data storage requirements, as well as a large increase in the speed of operation. You can achieve large performance gains on most processors by identifying those portions of your block diagram that are really integer calculations (such as accumulators), and implementing them with integer data types.

Floating-point DSP targets are an obvious exception to this rule.

The accumulator from the previous example used 64-bit floating-point calculations by default. The block diagram in the following figure implements the accumulator with 16-bit integer operations.



Accumulator Implemented with 16-bit Integers

If the **Saturate on integer overflow** option of the Sum block is turned off, the code generated from the integer implementation looks the same as code

generated from the floating-point block diagram. However, since `Sum_synth_accum` is performing integer arithmetic internally, the accumulator executes more efficiently.

Note that, by default, the **Saturate on integer overflow** option is on. This option generates extra error-checking code from the integer implementation, as in the following example.

```
static void accum_int16_output(int_T tid)
{
    /* local block i/o variables */

    int16_T rtb_UnitDelay_k;

    /* UnitDelay: '<Root>/Unit Delay' incorporates:
     * Constant: '<Root>/Constant'
     *
     * Regarding '<Root>/Unit Delay':
     * Operating as an accumulator
     */
    {
        int16_T tmpVar = accum_int16_DWork.UnitDelay_DSTATE;
        accum_int16_DWork.UnitDelay_DSTATE = tmpVar + (1);
        if ((tmpVar >= 0) && ((1) >= 0) &&
            (accum_int16_DWork.UnitDelay_DSTATE < 0))
        {
            accum_int16_DWork.UnitDelay_DSTATE = MAX_int16_T;
        } else if ((tmpVar < 0) && ((1) < 0) &&
            (accum_int16_DWork.UnitDelay_DSTATE >= 0)) {
            accum_int16_DWork.UnitDelay_DSTATE = MIN_int16_T;
        }
    }
    rtb_UnitDelay_k = accum_int16_DWork.UnitDelay_DSTATE;

    /* Output: '<Root>/Out1' */
    accum_int16_Y.Out1 = rtb_UnitDelay_k;
}
```

The floating-point implementation would not have generated the saturation error checks, which apply only to integers. When using integer data types, consider whether or not you need to generate saturation checking code.

If you are able to ignore saturation checks, turn **Saturate on integer overflow** off for the Sum block. The generated code then omits the preceding checks:

```
static void accum_int16_output(int_T tid)
{

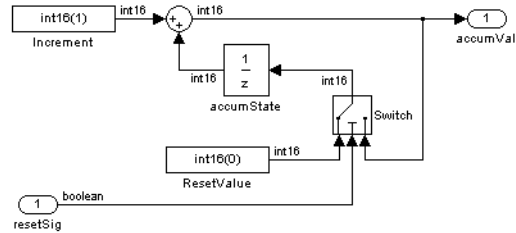
    /* local block i/o variables */

    int16_T rtb_UnitDelay_k;

    /* UnitDelay: '<Root>/Unit Delay' incorporates:
     * Constant: '<Root>/Constant'
     *
     * Regarding '<Root>/Unit Delay':
     * Operating as an accumulator
     */
    accum_int16_DWork.UnitDelay_DSTATE++;
    rtb_UnitDelay_k = accum_int16_DWork.UnitDelay_DSTATE;

    /* Outport: '<Root>/Out1' */
    accum_int16_Y.Out1 = rtb_UnitDelay_k;
}
```

The following figure shows an efficient way to add reset capability to the accumulator. When `resetSig` is greater than or equal to the threshold of the Switch block, the Switch block passes the reset value (0) back into the accumulator.



Integer Accumulator with Reset via External Input

The reset signal can protect computations from overflows, and the size of the resultant code is minimal. The code uses no floating-point operations.

```
static void accum_rst_output(int_T tid)
{
    /* local block i/o variables */

    int16_T rtb_Sum_j;

    /* Sum: '<Root>/Sum' */
    rtb_Sum_j = (int16_T)(1 + accum_rst_DWork.accumState_DSTATE);

    /* Outport: '<Root>/accumVal' */
    accum_rst_Y.accumVal = rtb_Sum_j;

    /* Switch: '<Root>/Switch' */
    if(accum_rst_U.resetSig) {
        accum_rst_B.Switch_k = 0;
    } else {
        accum_rst_B.Switch_k = rtb_Sum_j;
    }
}
```

This example uses an input to the system as the reset value, but you can also use an int16 constant.

Additional Integer and Fixed-Point Optimizations

You may find several companion products useful in optimizing the performance and size of integer-based generated code.

Generating Pure Integer Code with Real-Time Workshop Embedded Coder

The Real-Time Workshop Embedded Coder target provides the **Support floating point numbers** option (formerly described as the **Integer code only** option) to control whether generated code contains any floating-point data or operations. When this option is deselected, an error is raised if any noninteger data or expressions are encountered during compilation of the model. The error message reports the offending blocks and parameters.

If pure integer code generation is important to your design, you should consider using the Real-Time Workshop Embedded Coder target (or a target of your own, based on the Real-Time Workshop Embedded Coder target).

The Real-Time Workshop Embedded Coder target offers many additional optimizations. See the Real-Time Workshop Embedded Coder documentation for further information.

Optimizing Integer Code with Simulink Fixed Point and Stateflow

Simulink Fixed Point (a separate product) is designed to deliver the highest levels of performance for noninteger algorithms on processors lacking floating-point hardware. Simulink Fixed Point code generation in Real-Time Workshop implements calculations using a processor's integer operations. The code generation strategy maps the integer value set to a range of expected real-world values to achieve the high efficiency.

Finite-state machine or flowchart constructs can often represent decision logic (or mode logic) efficiently. Stateflow (a separate product) provides these capabilities. Stateflow, which is fully integrated into Simulink, supports integer data-typed code generation.

Writing S-Functions for Real-Time Workshop

The following sections explain how to write S-functions that work with Real-Time Workshop.

Introduction (p. 10-2)	Describes various approaches to writing S-functions for Real-Time Workshop
Noninlined S-Functions (p. 10-7)	Explains the noninlined approach to writing S-functions for Real-Time Workshop
Writing Wrapper S-Functions (p. 10-9)	Creating S-functions that serve as wrappers for existing code
Fully Inlined S-Functions (p. 10-18)	Explains the inlined approach to writing S-functions for Real-Time Workshop
Fully Inlined S-Function with the mdlRTW Routine (p. 10-20)	How to use the mdlRTW callback method in an inlined S-function
Supporting Expression Folding in S-Functions (p. 10-46)	How to realize the efficiencies offered by expression folding at input and output ports in inlined S-functions
Creating Code-Reuse-Compatible S-Functions (p. 10-61)	How to create S-functions that are compatible with the Real-Time Workshop subsystem code reuse feature
Coding S-functions for Multirate Multitasking Environments (p. 10-62)	How to handle rate-grouped tasks in multirate, port-based sample time S-functions

Introduction

This chapter describes how to create S-functions that work seamlessly with Real-Time Workshop. It begins with basic concepts and concludes with an example of how to create a highly optimized direct-index lookup table S-Function block.

This chapter assumes that you understand these concepts:

- Level 2 S-functions
- Target Language Compiler (TLC) scripting
- How Real-Time Workshop creates generated code

See the Target Language Compiler Reference Guide and other portions of the Real-Time Workshop documentation for more detailed information about the code generation process.

Note When this chapter refers to actions performed by the Target Language Compiler, including parsing, caching, creating buffers, etc., the name Target Language Compiler is spelled out fully. When referring to code written in the Target Language Compiler syntax, this chapter uses the abbreviation TLC.

Note The guidelines presented in this chapter are for Real-Time Workshop users. Even if you do not currently use Real-Time Workshop, we recommend that you follow the practices presented in this chapter when writing S-functions, especially if you are creating general-purpose S-functions.

Classes of Problems Solved by S-Functions

S-functions help solve various kinds of problems you might face when working with Simulink and Real-Time Workshop. These problems include

- Extending the set of algorithms (blocks) provided by Simulink and Real-Time Workshop
- Interfacing legacy (hand-written) C-code with Simulink and Real-Time Workshop

- Generating highly optimized C-code for embedded systems

S-functions and S-function routines form an application program interface (API) that allows you to implement generic algorithms in the Simulink environment with a great deal of flexibility. This flexibility cannot always be maintained when you use S-functions with Real-Time Workshop. For example, it is not possible to access the MATLAB workspace from an S-function that is used with Real-Time Workshop. However, using the techniques presented in this chapter, you can create S-functions for most applications that work with the generated code from Real-Time Workshop.

Although S-functions provide a generic and flexible solution for implementing complex algorithms in Simulink, they require significant 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 Real-Time Workshop to inline your S-functions.

Types of S-Functions

The implementation of S-functions changes based on your requirements. This chapter discusses the typical problems that you may face and how to create S-functions for applications that need to work with Simulink and Real-Time Workshop. These are some (informally defined) common situations:

- 1 “I’m not concerned with efficiency. I just want to write one version of my algorithm and have it work in Simulink and Real-Time Workshop automatically.”
- 2 “I have a lot of hand-written code that I need to interface. I want to call my function from Simulink and Real-Time Workshop in an efficient manner.”

or said another way:

“I want to create a block for my blockset that will be distributed throughout my organization. I’d like it to be very maintainable with efficient code. I’d like my algorithm to exist in one place but work with both Simulink and Real-Time Workshop.”

- 3 “I want to implement a highly optimized algorithm in Simulink and Real-Time Workshop that looks like a built-in block and generates very efficient code.”

The MathWorks has adopted terminology for these different requirements. Respectively, the situations described above map to this terminology:

- 1 Noninlined S-function
- 2 Wrapper S-function
- 3 Fully inlined S-function

Noninlined S-Functions

A noninlined S-function is a C-MEX S-function that is treated identically by Simulink and Real-Time Workshop. In general, you implement your algorithm once according to the S-function API. Simulink and Real-Time Workshop call the S-function routines (e.g., `mdlOutputs`) at the appropriate points during model execution.

Significant memory and computation resources are required for each instance of a noninlined S-Function block. However, this routine of incorporating algorithms into Simulink and Real-Time Workshop 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/or structures rapidly.

Note that writing a noninlined S-function does not involve any TLC coding. Noninlined S-functions are the default case for Real-Time Workshop in the sense that once you’ve built a C-MEX S-function in your model, there is no additional preparation prior to clicking **Build** in the **Real-Time Workshop** pane of the **Configuration parameters** dialog box for your model.

Wrapper S-Functions

A wrapper S-function is ideal for interfacing hand-written code or a large algorithm that is encapsulated within a few procedures. In this situation, usually the procedures reside in modules that are separate from the C-MEX S-function. The S-function module typically contains a few calls to your procedures. Because the S-function module does not contain any parts of your algorithm, but only calls your code, it is referred to as a *wrapper S-function*.

In addition to the C-MEX S-function wrapper, you need to create a TLC wrapper that complements your S-function. The TLC wrapper is similar to the S-function wrapper in that it contains calls to your algorithm.

Fully Inlined S-Functions

A fully inlined S-function builds your algorithm (block) into Simulink and Real-Time Workshop 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 Simulink (C-MEX S-function) and once for Real-Time Workshop (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.

Basic Files Required for Implementation

This section briefly describes what files and functions you'll need to create noninlined, wrapper, and fully inlined S-functions.

- Noninlined S-functions require the C-MEX S-function source code *sfunction.c*.
- Wrapper S-functions that inline a call to your algorithm (your C function) require an *sfunction.tlc* file.
- 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 at the end of this chapter contains two operating modes — one for evenly spaced x-data and one for unevenly spaced x-data.
 - Fully inlined S-functions might require the placement of the `mdlRTW` routine in your S-function MEX-file *sfunction.c*. The `mdlRTW` routine lets you place information in *model.rtw*, which is the file that is processed by the Target Language Compiler prior to executing *sfunction.tlc* when generating code. This is useful when you want to introduce nontunable parameters into your TLC file.

For S-functions to work correctly in the Simulink environment, a certain amount of overhead code is necessary. When Real-Time Workshop generates code from models that contain S-functions (without *sfunction.tlc* files), it embeds some of this overhead code in the generated C 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 directs Real-Time Workshop to eliminate all overhead code from the generated code. The Target Language Compiler, which is part of Real-Time Workshop, 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. In MathWorks terminology, *inline* means to specify a text string in place of the call to the general S-function API routines (e.g., `mdl0outputs`). For example, when we say that a TLC file is used to inline an S-function, we mean that the generated code contains the appropriate C code that would normally appear within the S-function routines and the S-function itself has been removed from the build process.

Noninlined S-Functions

Noninlined S-functions are identified by the *absence* of an `sfunction.tlc` file for your S-function (`sfunction.mex`). When you place a noninlined S-function in a model that is to be used with Real-Time Workshop, the following MATLAB API functions are supported:

- `mxGetEps`
- `mxGetInf`
- `mxGetM`
- `mxGetN`
- `mxGetNaN`
- `mxGetPr` — Note that using `mxGetPr` on an empty matrix does not return `NULL`; rather, it returns a random value. Therefore, you should protect calls to `mxGetPr` with `mxIsEmpty`.
- `mxGetScalar`
- `mxGetString`
- `mxIsEmpty`
- `mxIsFinite`
- `mxIsInf`

In addition, parameters to S-functions can only be of type double precision or characters contained in scalars, vectors, or 2-D matrices. To obtain more flexibility in the type of parameters you can supply to S-functions or the operations in the S-function, you need to inline your S-function and (possibly) use an `mdlRTW` S-function routine.

S-Function Module Names for Real-Time Workshop Builds

If your S-function is built with multiple modules, you must provide the build process names of additional modules. You can do this through Real-Time Workshop template makefile technology, or more conveniently by using the `set_param` MATLAB command. For example, if your S-function is built with multiple modules, as in

```
mex sfun_main.c sfun_module1.c sfun_module2.c
```

specify the names of the modules without the extension, using the command

```
set_param(sfun_block, 'SFunctionModules', 'sfun_module1 sfun_module2')
```

The parameter can also be a variable, as in

```
modules = 'sfun_module1 sfun_module2'  
set_param(sfun_block, 'SFunctionModules', modules)
```

or a string to be evaluated (this is needed when the modules are valid identifiers).

```
set_param(sfun_block, 'SFunctionModules', ''sfun_module1 sfun_module2'')
```


Writing Wrapper S-Functions

This section describes how to create S-functions that work seamlessly with Simulink and Real-Time Workshop using the *wrapper* concept. This section begins by describing how to interface your algorithms in Simulink by writing MEX S-function wrappers (*sfunction.mex*). It finishes with a description of how to direct Real-Time Workshop 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 your C code algorithms in Simulink and Real-Time Workshop with little or no change to your original C code function. A *MEX S-function wrapper* is an S-function that calls code that resides in another module. In effect, the wrapper binds your code to Simulink. A *TLC S-function wrapper* is a TLC file that specifies how Real-Time Workshop should call your code (the same code that was called from the C-MEX S-function wrapper).

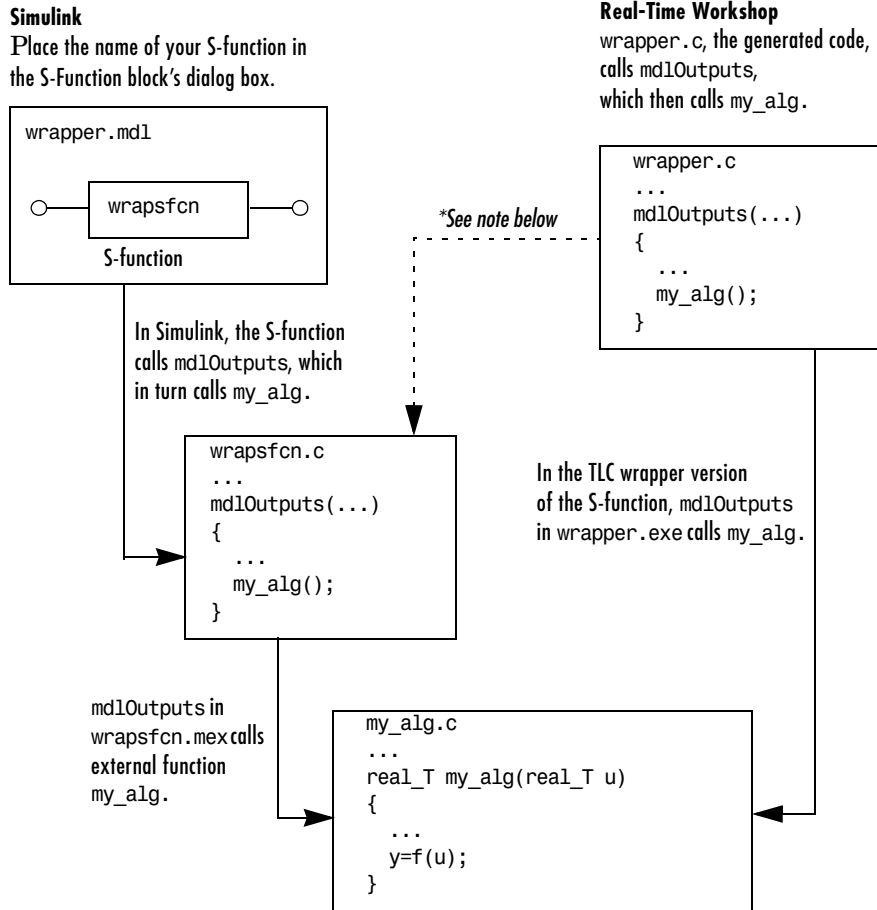
Suppose you have an algorithm (i.e., a C function) called `my_alg` that resides in the file `my_alg.c`. You can integrate `my_alg` into Simulink by creating a MEX S-function wrapper (e.g., `wrapsfcn.c`). Once this is done, Simulink can call `my_alg` from an S-Function block. However, the Simulink S-function contains a set of empty functions that Simulink requires for various API-related purposes. For example, although only `mdlOutputs` calls `my_alg`, Simulink calls `mdlTerminate` as well, even though this S-function routine performs no action.

You can integrate `my_alg` into Real-Time Workshop generated code (i.e., embed the call to `my_alg` in the generated code) by creating a TLC S-function wrapper (e.g., `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 Simulink. However, if you want to create code that is

- Interpretive in nature in Simulink (i.e., highly parameterized by operating modes)
- Heavily optimized in Real-Time Workshop (i.e., 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.
 This figure illustrates the wrapper S-function concept.

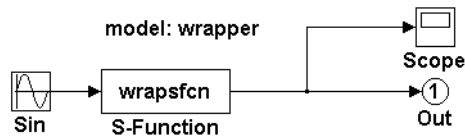


*The dotted line above is the path taken if the S-function does not have a TLC wrapper file. If there is no TLC wrapper file, the generated code calls md1Outputs.

How S-Functions Interface with Hand-Written Code

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 code algorithms from mdlOutputs. S-function wrappers have the advantage that you can quickly integrate large stand-alone C code into your model without having to make changes to the code.

This is an example of a model that includes an S-function wrapper.



An Example Model That Includes an S-Function Wrapper

There are two files associated with the wrapsfcn block, the S-function wrapper and the C code that contains the algorithm. This is the S-function wrapper code for this example, called wrapsfcn.c.

```

#define S_FUNCTION_NAME wrapsfcn
#define S_FUNCTION_LEVEL 2
#include "simstruc.h"

Declare my_alg as extern. { extern real_T my_alg(real_T u);

/*
 * 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]);
}
/*
 * 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_sfuns.h" /* Code generation registration function */
#endif

```

Place the call to my_alg in
mdlOutputs.



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:

```

#include "tmwtypes.h"
real_T my_alg(real_T u)
{
    return(u * 2.0);
}

```

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 section 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 have Real-Time Workshop embed the same type of call in the generated code.

It is instructive to look at how Real-Time Workshop 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, Real-Time Workshop generates 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')
```

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)
{
    /* (no start code 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);
```

Noninlined S-functions create a SimStruct object and generate a call to the S-function routine mdlOutputs.

```

/* Level2 S-Function Block: <Root>/S-Function (wrapsfcn) */
{
    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)
{
    /* (no update code required) */
}

/* Terminate function */
void mdlTerminate(void)
{
    /* Level2 S-Function Block: <Root>/S-Function (wrapsfcn) */
    {
        SimStruct *rts = ssGetSFunction(rtS, 0);
        sfcnTerminate(rts);
    }
}

#include "wrapper.reg"

/* [EOF] wrapper.c */

```

Noninlined S-functions require a SimStruct object and the call to the S-function routine mdlTerminate.

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:

```

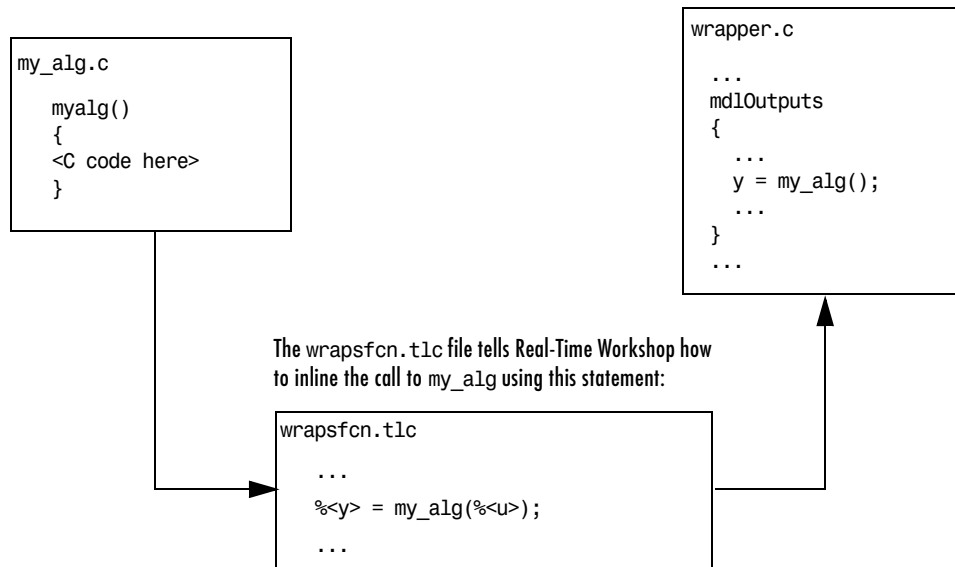
SimStruct *rts = ssGetSFunction(rtS, 0);
sfcnOutputs(rts, tid);

```

This call has a significant amount of computational overhead associated with it. First, Simulink creates a `SimStruct` data structure for the S-Function block. Second, Real-Time Workshop constructs a call through a function pointer to execute `mdlOutputs`, then `mdlOutputs` calls `my_alg`. By inlining the call to your

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; this file must contain the function call to `my_alg`. This picture shows the relationships between the algorithm, the wrapper S-function, and the `sfunction.tlc` file.



Inlining an Algorithm by Using a 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);
    %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"
    %<y> = my_alg(%<u>);
%endfunction %% Outputs

```

This line is placed in `wrapper.h`.

This line is expanded and placed in `mdlOutputs` within `wrapper.c`.

The first section of this code directs Real-Time Workshop to inline the `wrapsfcn` S-Function block and generate the code in C:

```

implements "wrapsfcn" "C"

```

The next task is to tell Real-Time Workshop 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 load the 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 no longer contains a call to an empty function and the `mdlOutputs` function now directly calls `my_alg`.

Inlined call to the function
`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);

        /* Outport Block: <Root>/Out */
        rtY.Out = rtB.S_Function;
    }
}
```

In addition, `wrapper.reg` no longer creates a child `SimStruct` for the S-function because the generated code is calling `my_alg` directly. This eliminates over 1 K of memory usage.

Fully Inlined S-Functions

Continuing the example of the previous section, you could eliminate the call to `my_alg` entirely by specifying the explicit code (i.e., `2.0 * u`) in `wrapsfcn.tlc`. This is referred to as a *fully inlined S-function*. While this can improve performance, if your C code is large this can be a lengthy task. In addition, you now have to maintain your algorithm in two places, the 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;

    /* Output Block: <Root>/Out */
    rtY.Out = rtB.S_Function;
}
```

This is the explicit embedding
of the algorithm. {

The Target Language Compiler has replaced the call to `my_alg` with the algorithm itself.

Multiport S-Function Example

A more advanced multiport inlined S-function example exists in `matlabroot/simulink/src/sfun_multiport.c` and `matlabroot/toolbox/simulink/blocks/tlc_c/sfun_multiport.tlc`. This S-function demonstrates 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.

Fully Inlined S-Function with the mdlRTW Routine

You can make a more fully inlined S-function that uses 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, including

- Renaming of tunable parameters in the generated code. This improves readability of the code by replacing `p1`, `p2`, etc., by names of your choice.
- Creating a parameter record of a nontunable parameter for use with a TLC file.

`mdlRTW` does this by placing information in the `model.rtw` file. The `mdlRTW` routine is described in the text file `matlabroot/simulink/src/sfuntmpl_doc.c`.

As an example of how to use the `mdlRTW` function, this section 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 in Simulink, the first step is to write an S-function that executes the algorithm in `mdlOutputs`. To produce the most efficient C code, the next step is to create a corresponding TLC file to eliminate computational overhead and improve the performance of the lookup computations.

For your convenience, Simulink 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 section demonstrates how to create a 1-D lookup S-function, `sfun_directlook.c`, and its corresponding inlined `sfun_directlook.tlc` file. (See the Real-Time Workshop documentation and the Target Language Compiler Reference Guide documentation for more details on the Target Language Compiler.) This 1-D direct-index lookup table example demonstrates 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 routine to customize Real-Time Workshop generated code 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

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 strings 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(gcf, 'RTWdata', mydata)  
get_param(gcf, '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"  
    }  
}
```

The 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 is not needed in all situations. 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.

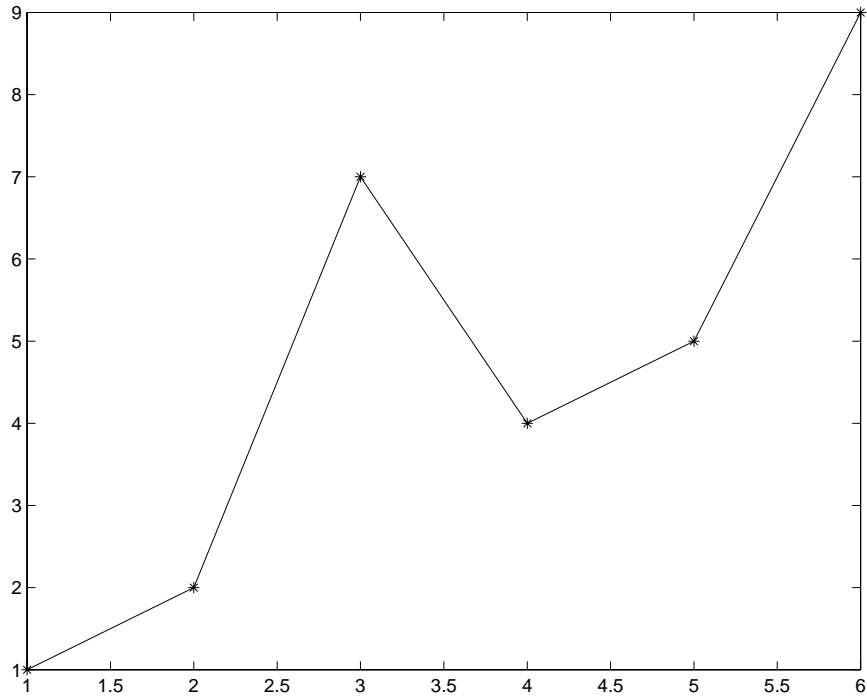
This direct 1-D lookup example computes an approximate solution $p(x)$ to a partially known function $f(x)$ at $x=x0$, given data point pairs (x,y) in the form of an x -data vector and a y -data vector. For a given data pair (e.g., the i th pair), $y_i = f(x_i)$. It is assumed that the x -data values 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, more efficient code is generated.

The following graph illustrates 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.



Typical Output from a Lookup Table Example

The Direct-Index Lookup Table Example

This section shows how to improve the lookup table by inlining a direct-index S-function with a TLC file. Note that this direct-index lookup table S-function does not require a TLC file to work with Real-Time Workshop. 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` routine 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 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 demonstrates 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 code, is `sfun_directlook.tlc` (see listing below).

Error Handling

In this example, the `mdlCheckParameters` routine verifies that

- The new parameter settings are correct.
- `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 if needed.

Note that the `mdlInitializeSizes` routine explicitly calls `mdlCheckParameters` after it verifies that the number of parameters passed to the S-function is correct. After Simulink 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 illustrates 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 line

```
ssSetSFcnParamTunable(S, iParam, SS_PRM_NOT_TUNABLE);
```

in `mdlInitializeSizes` tells Simulink to disallow changes to the `XDataEvenlySpaced` parameter. During execution, `mdlOutputs` accesses the value of `XDataEvenlySpaced` from `UserData` rather than calling the `mxGetPr` MATLAB API function. This results in a slight increase in performance.

mdlRTW Usage

Real-Time Workshop calls the mdlRTW routine while it (Real-Time Workshop) generates 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.

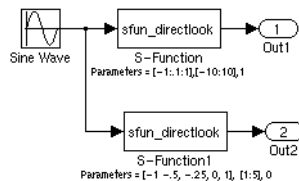
This example adds the following information to the *model.rtw* file:

- Parameter settings — These 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`). This example writes it out as a parameter setting (`XSpacing`) using the function `ssWriteRTWParamSettings`.

Note that because `xData` and `yData` are registered as run-time parameters in `mdlSetWorkWidths`, Real-Time Workshop handles writing to the *model.rtw* file automatically. See the section on run-time parameters in the Simulink documentation.

Example Model

Before examining the S-function and the inlined TLC file, consider the generated code for the following model.



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 Real-Time Workshop build process that the module `lookup_index.c` is needed when creating the executable.

The generated *model.c* code for the lookup table example model is

```

/*
 * sfun_directlook_ex.c
 *
 * Real-Time Workshop code generation for Simulink model "sfun_directlook_ex.mdl".
 *
 * Model Version          : 1.2
 * Real-Time Workshop version : 6.0 (R14 Prerelease 2) 06-Apr-2004
 * C source code generated on : Fri Apr 09 09:15:12 2004
 */

#include "sfun_directlook_ex.h"
#include "sfun_directlook_ex_private.h"

/* External output (root outputs fed by signals with auto storage) */
ExternalOutputs_sfun_directlook_ex sfun_directlook_ex_Y;

/* Real-time model */
rtModel_sfun_directlook_ex sfun_directlook_ex_M;
rtModel_sfun_directlook_ex *sfun_directlook_ex_M = &sfun_directlook_ex_M;

/* Model output function */
static void sfun_directlook_ex_output(int_T tid)
{
    /* local block i/o variables */

    real_T rtb_SFunction_h;
    real_T rtb_temp1;

    /* Sin: '<Root>/Sine Wave' */
    rtb_temp1 = sfun_directlook_ex_P.SineWave_Amp *
        sin(sfun_directlook_ex_P.SineWave_Freq * sfun_directlook_ex_M->Timing.t[0] +
            sfun_directlook_ex_P.SineWave_Phase) + sfun_directlook_ex_P.SineWave_Bias;

    /* S-Function Block: <Root>/S-Function */
    {
        const real_T *xData = &sfun_directlook_ex_P.SFunction_XData[0];
        const real_T *yData = &sfun_directlook_ex_P.SFunction_YData[0];
        real_T spacing = xData[1] - xData[0];

        if ( rtb_temp1 <= xData[0] ) {
            rtb_SFunction_h = yData[0];
        } else if ( rtb_temp1 >= yData[20] ) {
            rtb_SFunction_h = yData[20];
        } else {
            int_T idx = (int_T)( ( rtb_temp1 - xData[0] ) / spacing );
            rtb_SFunction_h = yData[idx];
        }
    }
}

```

This is the code that is inlined for the top S-Function block in the sfun_directlook_ex model.

This is the code that is inlined for the bottom S-Function block in the `sfun_directlook_ex` model.

```

    }
}

/* Outport: '<Root>/Out1' */
sfun_directlook_ex_Y.Out1 = rtb_SFunction_h;

/* S-Function Block: <Root>/S-Function1 */
{
    const real_T *xData = &sfun_directlook_ex_P.SFunction1_XData[0];
    const real_T *yData = &sfun_directlook_ex_P.SFunction1_YData[0];
    int_T idx;

    idx = GetDirectLookupIndex(xData, 5, rtb_temp1);
    rtb_temp1 = yData[idx];
}

/* Outport: '<Root>/Out2' */
sfun_directlook_ex_Y.Out2 = rtb_temp1;
}

/* Model update function */
static void sfun_directlook_ex_update(int_T tid)
{
    /* Update absolute time for base rate */

    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 * 0x10000;

    {
        /* Update absolute timer for sample time: [0.1s, 0.0s] */

        if(!(++sfun_directlook_ex_M->Timing.clockTick1))
            ++sfun_directlook_ex_M->Timing.clockTickH1;
        sfun_directlook_ex_M->Timing.t[1] = sfun_directlook_ex_M->Timing.clockTick1
            * sfun_directlook_ex_M->Timing.stepSize1 +
            sfun_directlook_ex_M->Timing.clockTickH1 *
            sfun_directlook_ex_M->Timing.stepSize1 * 0x10000;
    }
}

/* Model initialize function */
void sfun_directlook_ex_initialize(boolean_T firstTime)
{
    if (firstTime) {
        /* registration code */
        /* initialize real-time model */
    }
}

```

```

(void)memset((char_T *)sfun_directlook_ex_M, 0,
sizeof(rtModel_sfundirectlook_ex));

{
/* Setup solver object */

rtsiSetSimTimeStepPtr(&sfun_directlook_ex_M->solverInfo,
&sfun_directlook_ex_M->Timing.simTimeStep);
rtsiSetTPtr(&sfun_directlook_ex_M->solverInfo,
&rtmGetTPtr(sfun_directlook_ex_M));
rtsiSetStepSizePtr(&sfun_directlook_ex_M->solverInfo,
&sfun_directlook_ex_M->Timing.stepSize0);
rtsiSetErrorStatusPtr(&sfun_directlook_ex_M->solverInfo,
&rtmGetErrorStatus(sfun_directlook_ex_M));

rtsiSetRTModelPtr(&sfun_directlook_ex_M->solverInfo, sfun_directlook_ex_M);
}
rtsiSetSimTimeStep(&sfun_directlook_ex_M->solverInfo, MAJOR_TIME_STEP);

/* Initialize timing info */
{
int_T *mdlTsMap = sfun_directlook_ex_M->Timing.sampleTimeTaskIDArray;
int_T i;

for(i = 0; i < 2; i++) {
mdlTsMap[i] = i;
}
sfun_directlook_ex_M->Timing.sampleTimeTaskIDPtr = (&mdlTsMap[0]);
sfun_directlook_ex_M->Timing.sampleTimes =
(&sfun_directlook_ex_M->Timing.sampleTimesArray[0]);
sfun_directlook_ex_M->Timing.offsetTimes =
(&sfun_directlook_ex_M->Timing.offsetTimesArray[0]);
/* task periods */
sfun_directlook_ex_M->Timing.sampleTimes[0] = (0.0);
sfun_directlook_ex_M->Timing.sampleTimes[1] = (0.1);

/* task offsets */
sfun_directlook_ex_M->Timing.offsetTimes[0] = (0.0);
sfun_directlook_ex_M->Timing.offsetTimes[1] = (0.0);
}

rtmSetTPtr(sfun_directlook_ex_M, &sfun_directlook_ex_M->Timing.tArray[0]);

{
int_T *mdlSampleHits = sfun_directlook_ex_M->Timing.sampleHitArray;
int_T i;

for(i = 0; i < 2; i++) {
mdlSampleHits[i] = 1;
}
sfun_directlook_ex_M->Timing.sampleHits = (&mdlSampleHits[0]);
}

```

```

rtmSetTFinal(sfundirectlook_ex_M, 10.0);
sfundirectlook_ex_M->Timing.stepSize0 = 0.1;
sfundirectlook_ex_M->Timing.stepSize1 = 0.1;

/* Setup for data logging */
{
    static RTWLogInfo rt_DataLoggingInfo;

    sfundirectlook_ex_M->rtwLogInfo = &rt_DataLoggingInfo;

    rtliSetLogFormat(sfundirectlook_ex_M->rtwLogInfo, 0);

    rtliSetLogMaxRows(sfundirectlook_ex_M->rtwLogInfo, 1000);

    rtliSetLogDecimation(sfundirectlook_ex_M->rtwLogInfo, 1);

    rtliSetLogVarNameModifier(sfundirectlook_ex_M->rtwLogInfo, "rt_");

    rtliSetLogT(sfundirectlook_ex_M->rtwLogInfo, "tout");

    rtliSetLogX(sfundirectlook_ex_M->rtwLogInfo, "");

    rtliSetLogXFinal(sfundirectlook_ex_M->rtwLogInfo, "");

    rtliSetLogXSignalInfo(sfundirectlook_ex_M->rtwLogInfo, NULL);

    rtliSetLogXSignalPtrs(sfundirectlook_ex_M->rtwLogInfo, NULL);

    rtliSetLogY(sfundirectlook_ex_M->rtwLogInfo, "yout");

    /*
     * Set pointers to the data and signal info for each output
     */
    {
        static void * rt_LoggedOutputSignalPtrs[] = {
            &sfundirectlook_ex_Y.Out1,
            &sfundirectlook_ex_Y.Out2
        };

        rtliSetLogYSignalPtrs(sfundirectlook_ex_M->rtwLogInfo,
            ((LogSignalPtrsType)rt_LoggedOutputSignalPtrs));
    }

    {
        static int_T rt_LoggedOutputWidths[] = {
            1,
            1
        };

        static int_T rt_LoggedOutputNumDimensions[] = {
            1,
            1
        };
    }
}

```

```
static int_T rt_LoggedOutputDimensions[] = {
    1,
    1
};

static BuiltInDTypeId rt_LoggedOutputDataTypeIds[] = {
    SS_DOUBLE,
    SS_DOUBLE
};

static int_T rt_LoggedOutputComplexSignals[] = {
    0,
    0
};

static const char_T *rt_LoggedOutputLabels[] = {
    "",
    ""
};

static const char_T *rt_LoggedOutputBlockNames[] = {
    "sfun_directlook_ex/Out1",
    "sfun_directlook_ex/Out2"
};

static RTWLogDataTypeConvert rt_RTWLogDataTypeConvert[] = {
    { 0, SS_DOUBLE, SS_DOUBLE, 1.0, 0, 0.0},
    { 0, SS_DOUBLE, SS_DOUBLE, 1.0, 0, 0.0}
};

static RTWLogSignalInfo rt_LoggedOutputSignalInfo[] = {
    {
        2,
        rt_LoggedOutputWidths,
        rt_LoggedOutputNumDimensions,
        rt_LoggedOutputDimensions,
        rt_LoggedOutputDataTypeIds,
        rt_LoggedOutputComplexSignals,
        NULL,
        rt_LoggedOutputLabels,
        NULL,
        NULL,
        NULL,
        rt_LoggedOutputBlockNames,
        NULL,
        rt_RTWLogDataTypeConvert
    }
};

rtliSetLogYSignalInfo(sfun_directlook_ex_M->rtwLogInfo,
    rt_LoggedOutputSignalInfo);
}
}
```

```

sfun_directlook_ex_M->solverInfoPtr = (&sfun_directlook_ex_M->solverInfo);
sfun_directlook_ex_M->Timing.stepSize = (0.1);
rtsiSetFixedStepSize(&sfun_directlook_ex_M->solverInfo, 0.1);
rtsiSetSolverMode(&sfun_directlook_ex_M->solverInfo,
  SOLVER_MODE_SINGLETASKING);

/* parameters */
sfun_directlook_ex_M->ModelData.defaultParam = ((real_T *)
  &sfun_directlook_ex_P);

/* external outputs */
sfun_directlook_ex_M->ModelData.outputs = (&sfun_directlook_ex_Y);
sfun_directlook_ex_Y.Out1 = 0.0;
sfun_directlook_ex_Y.Out2 = 0.0;

/* initialize non-finites */
rt_InitInfAndNaN(sizeof(real_T));
}
}

/* Model terminate function */
void sfun_directlook_ex_terminate(void)
{
}

/*=====
 * Start of GRT compatible call interface
 *=====*/

void MdlOutputs(int_T tid) {
  sfun_directlook_ex_output(tid);
}

void MdlUpdate(int_T tid) {
  sfun_directlook_ex_update(tid);
}

void MdlInitializeSizes(void) {
  sfun_directlook_ex_M->Sizes.numContStates = (0); /* Number of continuous states */
  sfun_directlook_ex_M->Sizes.numY = (2); /* Number of model outputs */
  sfun_directlook_ex_M->Sizes.numU = (0); /* Number of model inputs */
  sfun_directlook_ex_M->Sizes.sysDirFeedThru = (0); /* The model is not direct
feedthrough */
  sfun_directlook_ex_M->Sizes.numSampTimes = (2); /* Number of sample times */
  sfun_directlook_ex_M->Sizes.numBlocks = (5); /* Number of blocks */
  sfun_directlook_ex_M->Sizes.numBlockIO = (0); /* Number of block outputs */
  sfun_directlook_ex_M->Sizes.numBlockPrms = (56); /* Sum of parameter "widths" */
}

void MdlInitializeSampleTimes(void) {

```

```
    }

    void MdlStart(void) {
    }

    rtModel_sfun_directlook_ex *sfun_directlook_ex(void) {
        sfun_directlook_ex_initialize(1);
        return sfun_directlook_ex_M;
    }

    void MdlTerminate(void) {
        sfun_directlook_ex_terminate();
    }

    /*=====
    * End of GRT compatible call interface
    *=====*/
```

Note that function names and global identifiers are prefixed with the model name. The final section of code, the GRT compatible call interface, provides wrapper functions to enable function calls to be compatible with the generic names used in code generated for GRT targets by prior versions of Real-Time Workshop.

matlabroot/simulink/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. No
 * interpolation is 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 1990-2004 The MathWorks, Inc.
 * $Revision: 1.15.4.3 $
 */

#define S_FUNCTION_NAME  sfun_directlook
#define S_FUNCTION_LEVEL 2

#include <math.h>
#include "simstruc.h"
#include <float.h>

/* use utility function IsRealVect() */
#ifdef(MATLAB_MEX_FILE)
#include "sfun_slutils.h"
#endif

/*=====
 * Build checking *
 *=====*/
#ifdef(MATLAB_MEX_FILE)
/*
 * This file cannot be used directly with the Real-Time Workshop. However,
 * this S-function does work with the Real-Time Workshop via
 * the Target Language Compiler technology. See
 * matlabroot/toolbox/simulink/blocks/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 that the new parameter setting are correct.
*
*   You should add a call to this routine from mdlInitializeSizes
*   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.
    */
    {
        int_T    i;
        boolean_T spacingEqual;
        real_T    *xData = mxGetPr(XVECT(S));

```

```

int_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, etc.).
 */
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
    *   i.e., 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);
    int_T          ny      = ssGetOutputPortWidth(S,0);
    int_T          xLen    = mxGetNumberOfElements(XVECT(S));
    int_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 the Real-Time Workshop 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 RTW. You will find
 *   the sfun_directlook.tlc in <matlaroot>/toolbox/simulink/blocks/tlc_c/.
 */
static void mdlRTW(SimStruct *S)
{
  /*
   * Write out the spacing setting as a param setting, i.e., 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/simulink/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 1990-2004 The MathWorks, Inc.
 * $Revision: 1.7 $
 */
#include "tmwtypes.h"

/*
 * 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 appropriate 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, i.e.,
                * 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, i.e.,
                * 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/blocks/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 1990-2004 The MathWorks, Inc.
%% $Revision: 1.8 $

%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.
%%          No interpolation is 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
```

Supporting Expression Folding in S-Functions

This section 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 section assumes that you are familiar with:

- Writing inlined S-functions (see “Writing S-Functions” in the Simulink documentation)
- The Target Language Compiler (see the Target Language Compiler documentation)

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, Simulink determines whether or not it can honor that request, given the block’s context. For example, Simulink 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 there are multiple output destinations.

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 10–47).

The sections 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 output port
- The conditions under which Simulink will honor or deny such requests

To take advantage of expression folding in your S-functions, you need to understand when it is appropriate to request acceptance and generation of expressions for specific blocks. It is not necessary for you to understand the

algorithm by which Simulink 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 any 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 involves no 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.

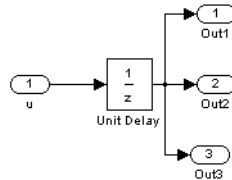


Diagram with Delay Block Routed to Multiple Destinations

The following code excerpt shows code generated from the Unit Delay block in this block diagram. Note that 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, involving no expressions, there is no performance penalty associated with using the trivial expression for multiple destinations.

```
void MdlOutputs(int_T tid)
{
    ...
    /* Output: <Root>/Out1 incorporates:
     *   UnitDelay: <Root>/Unit Delay */
    rtY.Out1 = rtDWork.Unit_Delay_DSTATE;

    /* Output: <Root>/Out2 incorporates:
     *   UnitDelay: <Root>/Unit Delay */
    rtY.Out2 = rtDWork.Unit_Delay_DSTATE;

    /* Output: <Root>/Out3 incorporates:
     *   UnitDelay: <Root>/Unit Delay */
    rtY.Out3 = rtDWork.Unit_Delay_DSTATE;

    ...
}
```

On the other hand, consider the Sum blocks in the model below:

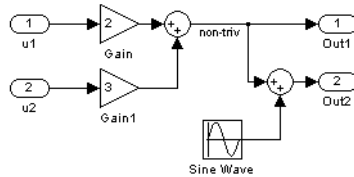


Diagram with Sum Block Routd to Multiple Destinations

The upper Sum block in Figure 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 of illustrates 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);
}
```

Specifying 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 ssSetOutputPortConstantOutputExprInRTW(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 any prior calls to the macros above:

- `bool ssGetOutputPortConstantOutputExprInRTW(SimStruct *S, int idx)`
- `bool ssGetOutputPortTrivialOutputExprInRTW(SimStruct *S, int idx)`
- `bool ssGetOutputPortOutputExprInRTW(SimStruct *S, int idx)`

Note that 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 section discusses block-specific conditions under which requests for input expressions are denied. For information on other conditions that prevent acceptance of expressions, see “Generic Conditions for Denial of Requests to Output Expressions” on page 10-55.

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 (e.g., 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 no block that is connected to that input port is permitted to output a generic or trivial expression.

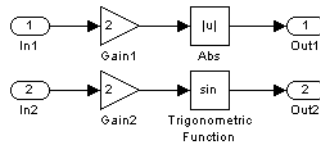
A request to output a constant expression is never denied, because there is no performance penalty for a constant expression, and it is always possible to take the parameter’s address.

Example: Acceptance and Denial of Expressions at Block Inputs

This example illustrates how various built-in blocks handle requests to accept different categories of expressions at their inputs.

The following sample model contains

- Two Gain blocks. Gain blocks request their destination blocks to accept generic expressions.
- An Abs block. This block always denies expressions at its input port. The Abs block code uses the macro `rt_ABS(u)`, which evaluates the input `u` twice. (See the TLC implementation of the Abs block in `matlabroot/rtw/c/tlc/blocks/absval.tlc`.)
- A Trigonometric Function block. This block accepts expressions at its input port.



Two Gain Blocks Requesting to Output an Expression

The Gain1 block's request to output an expression is denied by the Abs block. The Gain2 block's request to output an expression is accepted by the Trigonometric Function block.

The generated code is shown in the code excerpt below. Note that the output of the Gain1 block is stored in the temporary variable `rtb_Gain1`, rather than generating an input expression to the Abs block.

```
void MdlOutputs(int_T tid)
{
  /* local block i/o variables */
  real_T rtb_Gain1;

  /* Gain: '<Root>/Gain1' incorporates:
   *   Inport: '<Root>/In1'
   *
   *   Regarding '<Root>/Gain':
```

```
    *   Gain value: 2.0
    */
    rtb_Gain1 = rtU.In1 * 2.0;

    /* Output: '<Root>/Out1' incorporates:
    *   Abs: '<Root>/Abs'
    */
    rtY.Out1 = rt_ABS(rtb_Gain1);

    /* Output: '<Root>/Out2' incorporates:
    *   Trigonometry: '<Root>/Trigonometric Function'
    *   Gain: '<Root>/Gain2'
    *   Inport: '<Root>/In2'
    *
    *   Regarding '<Root>/Gain2':
    *   Gain value: 2.0
    */
    rtY.Out2 = sin((2.0 * rtU.In2));
}
```

Using the 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 (i.e., 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)
```

Generic Conditions for Denial of 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 (e.g. 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.

Utilizing Expression Folding in Your TLC Block Implementation

To take advantage of expression folding, an inlined S-Function must be modified in two ways:

- It must tell Simulink whether it generates or accepts expressions at its input ports, as described in “Using the S-Function API to Specify Input Expression Acceptance” on page 10-54.
- It must tell Simulink whether it generates or accepts expressions at its output ports, as described in “Categories of Output Expressions” on page 10-47.
- The TLC implementation of the block must be modified.

This section discusses required modifications to the TLC implementation.

Expression Folding Compliance

In the `BlockInstanceSetup` function of your S-function, you must ensure that your block registers that it is compliant with expression folding. If you fail to do this, any 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)
```

Note that you can conditionally disable expression folding at the inputs and outputs of a block by making the call to this function conditionally.

If you have overridden one of the TLC block implementations provided by Real-Time Workshop 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 sections.

Outputting 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,lc, 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
- `lc`: 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`: string 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 an appropriate text string for the output signal or address. The string 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 appropriate reference to the parameter.  This function *may*
%%      be used by Simulink when optimizing the Block IO data structure.
%%
%function BlockOutputSignal(block,system,portIdx,ucv,lcx,idx,retType) void
%switch retType
%case "Signal"
%return LibBlockParameter(Value,ucv,lcx,idx)
%case "SignalAddr"
%return LibBlockParameterAddr(Value,ucv,lcx,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,lcx,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 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,lc,idx,retType) void
    %switch retType
        %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
%roll sigIdx = RollRegions, lcv = RollThreshold, block, "Roller", rollVars
    %assign u0 = LibBlockInputSignal(0, ucv, lcv, idx)
    %assign u1 = LibBlockInputSignal(1, ucv, lcv, idx)
    %assign y0 = LibBlockOutputSignal(0, ucv, lcv, idx)
    %assign y1 = LibBlockOutputSignal(1, ucv, lcv, idx)
    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, all 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 string 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 string, suitable for a block comment, specifying the value of the block parameter.

```
%openfile commentBuf
$c(*) Gain value: %<LibBlockParameterForComment(Gain)>
%closefile commentBuf
%<LibCacheBlockComment(block, commentBuf)>
```

Creating Code-Reuse-Compatible S-Functions

The code reuse feature of Real-Time Workshop generates code for a subsystem in the form of a function that is invoked wherever the subsystem occurs in the model (see “Nonvirtual Subsystem Code Generation Options” on page 4-2). If a subsystem contains S-functions, the S-functions must be compatible with the code reuse feature. Otherwise, Real-Time Workshop might not generate reusable code from the subsystem or might generate incorrect code.

If you want your S-function to support the subsystem code reuse feature, you must ensure that the S-function meets the following requirements:

- The S-function must be inlined.
- Code generated from the S-function must not use static variables.
- The TLC code that generates the inlined S-function code must not use the `BlockInstanceData` function.
- 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.)

In addition to meeting the preceding requirements, your S-function must set the `SS_OPTION_WORKS_WITH_CODE_REUSE` flag (see `ssSetOptions`). This flag assures Real-Time Workshop that your S-function meets the requirements for subsystem code reuse.

Coding 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. Real-Time Workshop Embedded Coder generates 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” in the Real-Time Workshop Embedded Coder documentation.

For each multirate S-function that is not rate grouping-compliant, Real-Time Workshop issues the following warning when you build:

```
Warning: Real-Time Workshop: 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, as appropriate.

Creating Multitasking-Safe, 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. Note that frame support 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. Any 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 any 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 always safely 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 via `DWork`) when there is a special sample hit. If you do this, however, you must declare the output 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 always the value when both the faster and slower rate had their hits (and possible earlier input data as well, depending on the algorithm). Any 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: $y = u(\text{tslow})$
- Input: local and reusable
- Output: local and reusable
- `DirectFeedthrough`: yes

```
OutputFcn
if (ssIsSampleHit(".1")) {
    if (ssIsSpecialSampleHit("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: $y = \text{average}(u)$
- 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
       outputting. 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 any rate other than the one assigned to the input port. Similarly, the output should not be written to at any 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, only one DWork is needed. 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.

Note that in either case, the data that the fast rate computations operate on is always delayed, i.e., 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: $y = u(ts_{\text{slow}} - 1)$
- 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: $y = \sin(\text{tfast}) + u(\text{tslow}-1)$
- 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 will see it.*/
    DWork[0] = u;
}
```

The S-Function Target

S-functions are an important class of target for which Real-Time Workshop can generate code. The ability to encapsulate a subsystem into an S-function allows you to increase its execution efficiency and shield its internal logic from inspection and modification. The following sections describe the properties of S-function targets and demonstrate how to generate them. For further details on the structure of S-functions, see *Writing S-Functions* in the Simulink documentation.

Introduction (p. 11-2)	Overview of the S-function target and its applications
Creating an S-Function Block from a Subsystem (p. 11-4)	How to extract a subsystem from a model and use it to generate a reusable S-function component; a step-by-step demonstration
Tunable Parameters in Generated S-Functions (p. 11-10)	How to declare tunable parameters in generated S-functions and how they differ from those in other targets
Automated S-Function Generation (p. 11-12)	Step-by-step instructions for automatically generating an S-function from a subsystem
Restrictions (p. 11-16)	Limitations constraining the use of the S-function target
Unsupported Blocks (p. 11-19)	Blocks not supported by the S-function target
System Target File and Template Makefiles (p. 11-20)	Control files used by the S-function target

Introduction

Using the S-function target, you can build an S-function component and use it as an S-Function block in another model. The S-function code format 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 “Creating an S-Function Block from a Subsystem” on page 11-4 for an example of this procedure.
- Speeding up simulation. In many cases, 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 will continue to maintain its own unique data.

The S-function target generates noninlined S-functions. You can generate an executable from a model that contains generated S-functions by using the generic real-time or real-time malloc targets. You cannot use the Real-Time Workshop Embedded Coder target for this purpose, because it requires inlined S-functions.

You can place a generated S-Function block into another model from which you can generate another S-function format. This allows any level of nested S-functions.

Note that you should avoid nesting S-functions 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. Real-Time Workshop currently does not detect such loops in S-function dependency, which can result in aborting or hanging MATLAB.

To prevent this from happening, you should be sure to name the subsystem or model to be generated as an S-function target uniquely, to avoid duplicating any existing MEX filenames on the MATLAB path.

Intellectual Property Protection

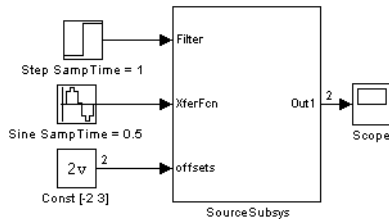
In addition to the technical applications of the S-function target listed above, you can use the S-function target to protect your designs and algorithms. By generating an S-function from a proprietary model or algorithm, you can share the model's functionality without providing the source code. You need only provide the binary .dll or MEX-file object to users.

Creating an S-Function Block from a Subsystem

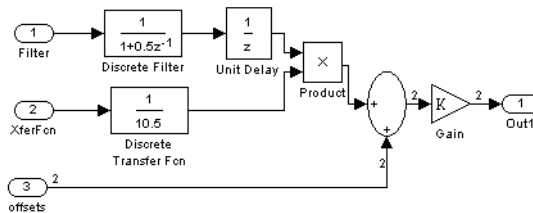
This section demonstrates how to extract a subsystem from a model and generate a reusable S-function component from it.

The following figure illustrates SourceModel, a simple model that inputs signals to a subsystem. The subsequent figure illustrates 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

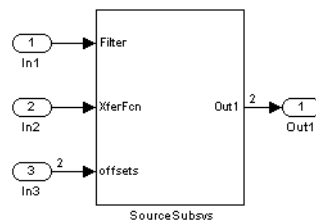
The objective is to extract SourceSubsys from the model and build an S-Function block from it, using the S-function target. The S-Function block must perform identically to the subsystem from which it was generated.

Note that 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 will have 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 11-8.)

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 of the original model. This ensures that the generated S-function component will operate identically to the original subsystem (see “Choice of Solver Type” on page 11-8 for an exception to this rule).

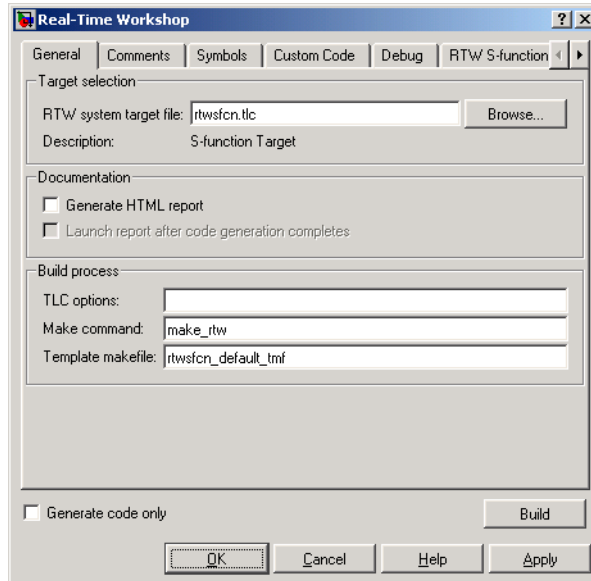
To build `SourceSubsys` as an S-function component,

- 1 Create a new model and copy/paste `SourceSubsys` 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 output. Connect inports and an output to `SourceSubsys`, as shown below.

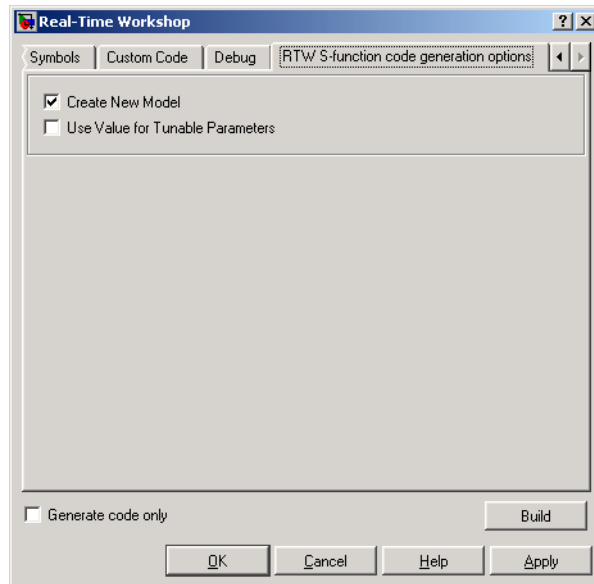


Note that the correct signal widths and sample times are propagated to these ports.

- 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 Save the new model.
- 6 In Model Explorer or the **Configuration Parameters** dialog, click the **Real-Time Workshop** tab.
- 7 Click the **Browse** button to open the System Target Browser. Select the S-function Target in the System Target Browser, and click **OK**. The **Real-Time Workshop** pane now appears as below:

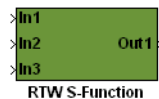


- 8 Select the **RTW S-function code generation options** tab (in Model Explorer) or node (in the **Configuration Parameters** dialog). Make sure that **Create New Model** is selected, as shown below:

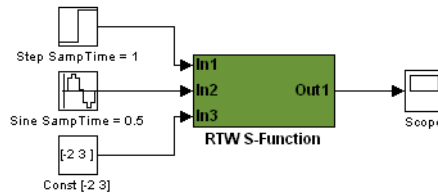


When this option is selected, the build process creates a new model after it builds the S-function component. The new model contains an S-Function block, linked to the S-function component.

- 9 Click **Apply** if necessary.
- 10 Click **Build**.
- 11 Real-Time Workshop builds the S-function component in the working directory. After the build, a new model window is displayed.



12 You can now copy the Real-Time Workshop S-Function block from the new model and use it in other models or in a library. The following figure shows the S-Function block plugged into the original model. Given identical input signals, the S-Function block will perform identically to the original subsystem.



Generated S-Function Configured Like SourceModel

Note that 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.

Sample Time Propagation in Generated S-Functions

Note that sample time propagation for the S-function code format is slightly different from the other code formats. A generated S-Function block will inherit its sample time from the model in which it is placed if (and only if) no blocks in the original model specify their sample times.

In addition, entering a specific **Fixed step size** (as opposed to auto) on the **Solver** pane will cause the generated S-function to not inherit a sample time.

Choice of Solver Type

If the model containing the subsystem from which you generate an S-function uses a variable-step solver, the generated S-function will contain zero crossing functions. Therefore, the generated S-function will work properly only in models that use variable-step solvers.

and more complicated models. By using generated S-functions, you can increase the efficiency of your modeling process.

Sample Time Propagation in Generated S-Functions

Note that sample time propagation for the S-function code format is slightly different from the other code formats. A generated S-Function block inherits its sample time from the model in which it is placed if (and only if) no blocks in the original model specify their sample times.

In addition, entering a specific **Fixed step size** (as opposed to auto) on the **Solver** pane causes the generated S-function not to inherit a sample time.

Choice of Solver Type

If the model containing the subsystem from which you generate an S-function uses a variable-step solver, the generated S-function will contain zero-crossing functions. Therefore, the generated S-function will work properly only in models that use variable-step solvers.

On the other hand, if the model containing the subsystem from which you generate an S-function uses a fixed-step solver, the generated S-function contains no zero-crossing functions. In this case, the generated S-function will work properly in models with either variable-step or fixed-step solvers.

Tunable Parameters in Generated S-Functions

You can use tunable parameters in generated S-functions in two ways:

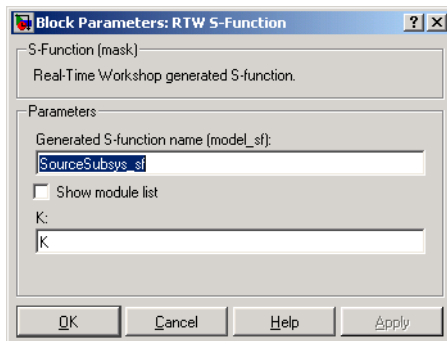
- Use the **Generate S-function** feature (see “Automated S-Function Generation” on page 11-12).
- or
- Use the **Model Parameter Configuration** dialog (see “Parameters: Storage, Interfacing, and Tuning” on page 5-2) to declare desired block parameters tunable.

Block parameters that are declared tunable with the auto storage class in the source model become tunable parameters of the generated S-function.

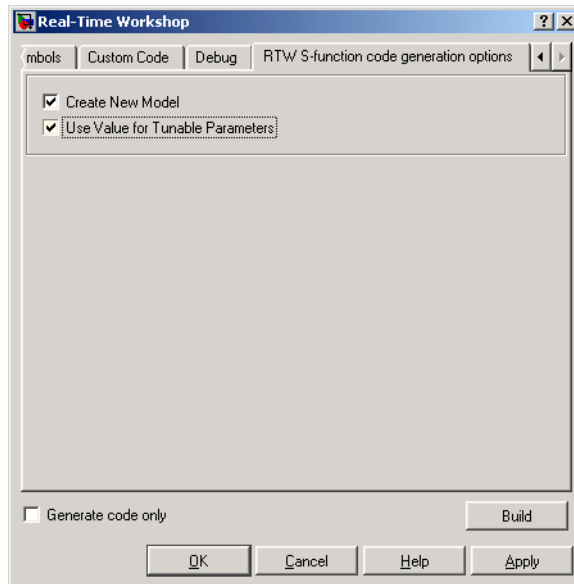
Note that these parameters do not become part of a generated *model_P* (formerly *rtP*) parameter data structure, as they would in code generated from other targets. Instead, the generated code accesses these parameters via MEX API calls such as `mxGetPr` or `mxGetData`. Your code should access these parameters in the same way.

For further information on MEX API calls, see “Writing S-Functions” and “External Interfaces/API” in the MATLAB online documentation.

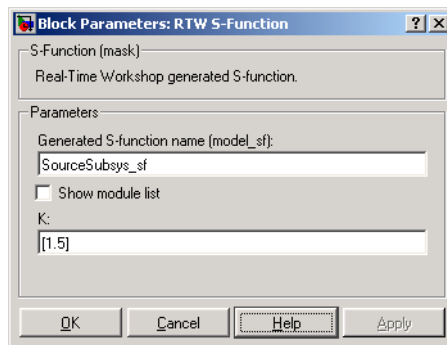
S-Function blocks created via the S-function target are automatically masked. The mask displays each tunable parameter in an edit field. By default, the edit field displays the parameter by variable name, as in the following example.



You can choose to display the value of the parameter rather than its variable name. To do this, select **Use Value for Tunable Parameters** in the **Options** section.



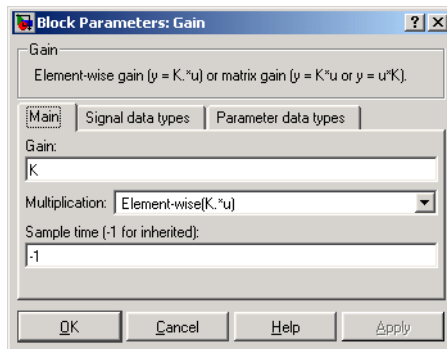
When this option is chosen, the value of the variable (at code generation time) is displayed in the edit field, as in the following example.



Automated S-Function Generation

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 subsystem illustrated below. 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 gain factor of the Gain block should be set within SourceSubsys to the workspace variable K (as illustrated below) and declare K as a tunable parameter.



To autogenerate an S-function from SourceSubsys with tunable parameter K,

- 1 Click the subsystem to select it.
- 2 Select **Generate S-function** from the **Real-Time Workshop** submenu of the **Tools** menu. This menu item is enabled when a subsystem is selected in the current model.

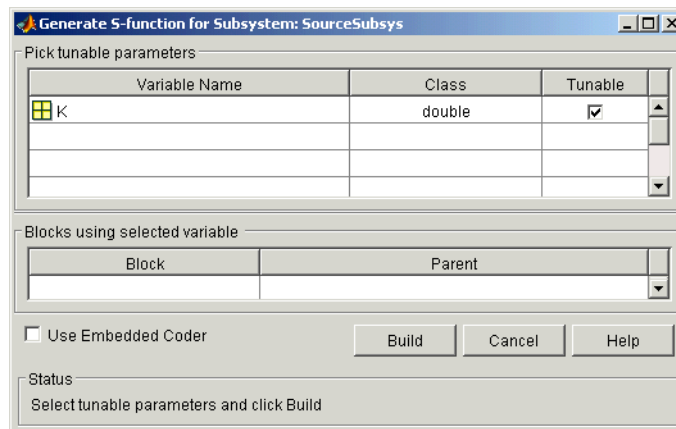
Alternatively, you can choose **Generate S-function** from the **Real-Time Workshop** submenu of the subsystem block's context menu.

- 3 The **Generate S-function window** is displayed (see the following figure). This window shows all 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 figure below, the parameter K is declared tunable.

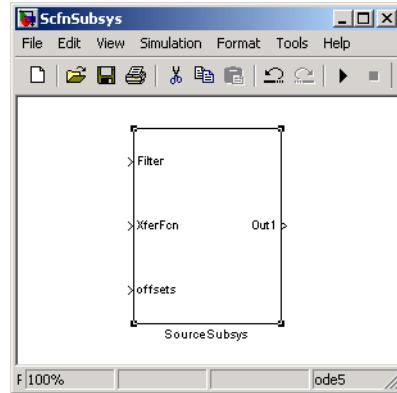
When you select a parameter in the upper pane, the lower pane shows all the blocks that reference the parameter, and the parent system of each such block.



The Generate S-Function Window

- 4 If you have installed the Real-Time Workshop Embedded Coder, the **Use Embedded Coder** check box is available, as shown above. Otherwise, it is grayed out. When **Use Embedded Coder** is selected, the build process generates a wrapper S-Function via the Real-Time Workshop Embedded Coder. See the Real-Time Workshop Embedded Coder documentation for further information.
- 5 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.

- 6 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.



- 7 The model window contains an S-Function block, *subsys*, where *subsys* is the name of the subsystem from which the block was generated.

The generated S-function component, *subsys* (which has been named *SourceSubsys* in the above example), is stored in the working directory. The generated source code for the S-function is written to a build directory, *subsys_sfnc_rtw*. Additionally a stub file, *subsys_sf.c*, is written to the working directory. This file simply contains an include directive that you can use to interface other C code to the generated code.

Note that if the **Use Embedded Coder** option is selected, the build directory is named *subsys_ert_rtw*.

- 8 Note that the untitled generated model does not persist unless you save it via the **File** menu, as shown in the previous picture.
- 9 Note that the generated S-Function block has inports and outports whose widths and sample times correspond to those of the original model.

The following code fragment, from the mdlOutputs routine of the generated S-function code (in SourceSubsys_sf.c), illustrates how the tunable variable K is referenced via 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))));
```

Note In automatic S-function generation, the **Use Value for Tunable Parameters** option is always set to its default value (off).

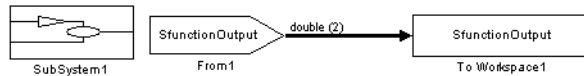
Restrictions

Limitations on Use of Goto and From Blocks

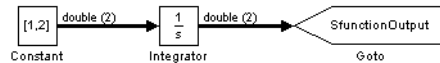
When using the S-function target, Real-Time Workshop restricts I/O to correspond to the root model's Inport and Outport blocks (or the Inport and Outport blocks of the Subsystem block from which the S-function target was generated). No code is generated for Goto or From blocks.

To work around this restriction, you should 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 needed Goto and From blocks.

As an example of this restriction, consider the model shown below, and its subsystem, Subsystem1, shown in the subsequent figure. The Goto block in Subsystem1, which has global visibility, passes its input to the From block in the root model.

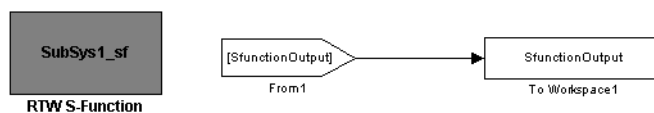


Root Model with From Block



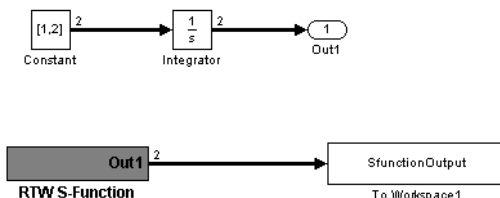
Subsystem1 with Goto Block

If SubSystem1 is built as an S-Function using the S-Function target, and plugged into the original model (as shown in the following figure), a warning is issued when the model is run, because the generated S-function does not implement the Goto block.



Generated S-Function Replaces Subsystem 1

A workaround is shown in the next figure. A conventional Outport block is used in Subsystem1. When the generated S-function is plugged into the root model, its output is connected to the To Workspace block.



Use of Outport in Generated S-Function

Model Reference and Updating Restrictions

The following restrictions apply to building and regenerating S-function targets.

- You cannot build models that contain Model blocks using the Real-Time Workshop S-function target. This also means that you cannot build a subsystem module by right-clicking (or via **Tools -> Real-Time Workshop -> Build subsystem**) if the subsystem contains Model blocks. This restriction applies only to GRT S-functions, not to ERT S-functions.
- If you modify the model that generated an S-Function block, Real-Time Workshop 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 options** configuration setting).

- Hand-written S-functions without corresponding TLC files must contain exception-free code. For more information on exception-free code, refer to “Exception-Free Code” in the Writing S-Functions Simulink documentation.

Unsupported Blocks

The S-function format does not support the following built-in blocks:

- MATLAB Fcn block
- S-Function blocks containing any of the following:
 - M-file S-functions
 - Fortran S-functions
 - C MEX S-functions that call into MATLAB
- Scope block
- To Workspace block

System Target File and Template Makefiles

The following system target file and template makefiles are provided for use with the S-function target.

System Target File

- `rtwsfcn.tlc`

Template Makefiles

- `rtwsfcn_bc.tmf` — Borland C
- `rtwsfcn_lcc.tmf` — LCC compiler
- `rtwsfc_unix.tmf` — UNIX host
- `rtwsfcn_vc.tmf` — Visual C
- `rtwsfcn_watc.tmf` — Watcom C

Real-Time Workshop

Rapid Simulation Target

The rapid simulation (RSim) target provides a fast and flexible platform on your host computer for testing code generated for models, tuning parameters, and varying inputs to compile statistics describing the behavior of your model across a range of initial conditions.

Introduction (p. 12-2)

Overview of the Rapid Simulation target, its applications, and dependencies on Simulink

Building for the Rapid Simulation Target (p. 12-5)

Generating, building, and running an RSim executable

Introduction

The Real-Time Workshop rapid simulation target (*RSim*) consists of a set of target files for non-real-time execution on your host computer. You can use RSim to generate fast, stand-alone simulations that allow batch parameter tuning and loading of new simulation data (signals) from a standard MATLAB MAT-file without needing to recompile your model.

The C code generated from Real-Time Workshop is highly optimized to provide fast execution of Simulink models of hybrid dynamic systems. This includes models using variable-step solvers and zero-crossing detection.

The speed of the generated code makes the RSim target ideal for batch or Monte Carlo simulation. The generated executable (*model.exe*) created using the RSim target has the necessary run-time interface to read and write data to standard MATLAB MAT-files. Using this interface *model.exe* can read new signals and parameters from input MAT-files at the start of the simulation and write the simulation results to output MAT-files.

Having built an RSim executable with Real-Time Workshop and an appropriate C compiler for your host computer, you can perform any combination of the following by using command-line options. Without recompiling, the rapid simulation target allows you to

- Specify new files to provide input signals for From File blocks
- Replace the entire block diagram parameter vector and run a simulation
- Specify a new stop time for ending the stand-alone simulation
- Specify a new name of the MAT-file used to save model output data
- Specify names of MAT-files used to save data connected to To File blocks

You can run these options:

- Directly from your operating system command line (for example, Windows Command Prompt or UNIX shell) or
- By using the bang (!) command with a command string at the MATLAB prompt

Therefore, you can easily write simple scripts that run a set of simulations in sequence while using new data sets. These scripts can provide unique filenames for both input parameters and input signals, as well as output filenames for the entire model or for To File blocks.

You can configure the RSim target to either access any solver available in Simulink or to use only the fixed-step solvers packaged with Real-Time Workshop. The default setting of `auto` for the **Solver** selection option in the **RSim Target** pane of the **Configuration Parameters** dialog configures RSim to use the Simulink solver module only when needed (i.e., when the model uses a variable-step solver).

In the default configuration, if the model uses a variable-step solver the stand-alone executable (*model* or *model.exe*) created by the RSim target links with the Simulink solver module (a shared library). When *model.exe* uses the Simulink solver module, running *model.exe* checks out a Simulink license (see details below). In such cases, *model.exe* requires read access to the installed location of MATLAB and Simulink in order to locate the `license.dat` file and the shared libraries.

Licensing Protocols for Simulink Solvers in Executables

The Rapid Simulation target supports variable-step solvers by linking the generated code with the Simulink solver module (a shared library). When this RSim executable is run, it accesses proprietary Simulink variable-step solver technology. In order to do so, the executable needs to check out a Simulink license for the duration of its execution.

Rapid Simulation executables that do not use the Simulink solver module (for example, RSim executables built for a fixed-step model using the Real-Time Workshop fixed-step solvers) do not require any license when they run.

The RSim executable looks in the default locations for the license file:

- UNIX: `matlabroot/etc/license.dat`
- PC: `matlabroot/bin/win32/license.dat`

Here, `matlabroot` is the one used when you built the RSim executable. If the RSim executable is unable to locate the license file (this can happen, for example, if you run this executable on another machine, where `matlabroot` is no longer valid), it displays the following error message and exits:

```
Error checking out SIMULINK license.
```

```
Cannot find license file
```

```
The license files (or server network addresses) attempted are
```

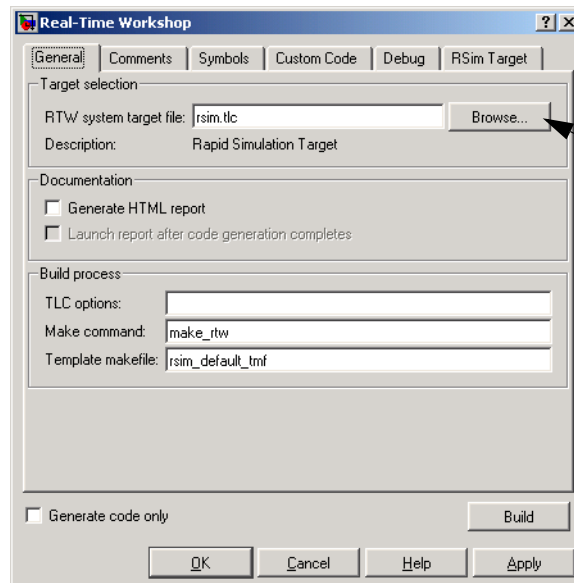
```
listed below. Use LM_LICENSE_FILE to use a different license
file, or contact your software provider for a license file.
Feature:          SIMULINK
Filename:         /apps/matlab/etc/license.dat
License path:    /abbs/matlab/etc/license.dat
FLEXlm error:    -1,359. System Error: 2 "No such file or directory"
For further information, refer to the FLEXlm End User Manual,
available at "www.globetrotter.com".
Error: Unable to checkout Simulink license
Error terminating RSim Engine: License check failed
```

Note You can point the RSim executable to a different license file by setting the environment variable `LM_LICENSE_FILE`. The location pointed to by that variable will override the default location compiled into the RSim executable.

If the RSim executable is unable to check out a Simulink license (this would happen, for example, if all Simulink licenses are currently checked out), or has other errors when checking out a Simulink license, it displays a detailed error message (similar to the one above) returned by the FLEXlm API and exits.

Building for the Rapid Simulation Target

To generate and build an RSim executable, press the **Browse** button on the **Real-Time Workshop** pane of the **Configuration Parameters** dialog or in the right-hand pane of Model Explorer (shown below). Then select the rapid simulation target from the **System Target File Browser**. This picture shows the target settings for the rapid simulation target.



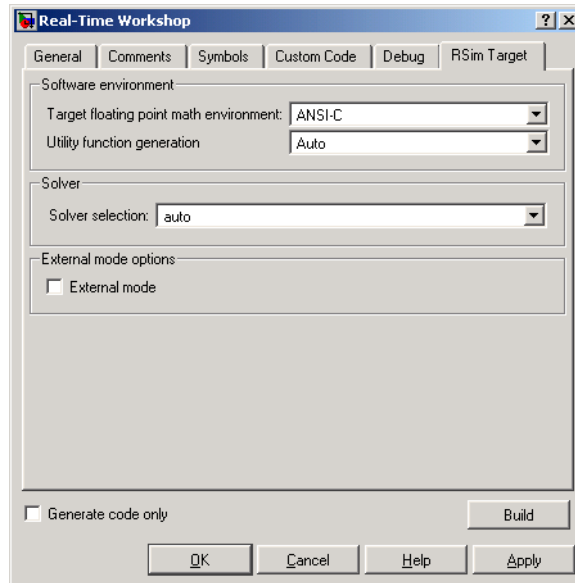
Click the **Browse** button and select the rapid simulation target from the System Target File Browser. This automatically selects the correct settings for the system target file, the template makefile, and the make command.

Specifying Target and Make Files for RSim

After specifying system target and make files as noted above, you might want to set up **Data Import/Export** and possibly External Mode communications. When you click **Build**, Real-Time Workshop automatically generates C code and build the RSim executable for your host machine using your host machine C compiler. See “Supported Compilers” on page 1-14, “Choosing and Configuring Your Compiler” on page 2-68, and “Template Makefiles and Make Options” on page 2-71 for additional information on compilers that are compatible with Simulink and Real-Time Workshop.

Configuration Options for RSim Targets

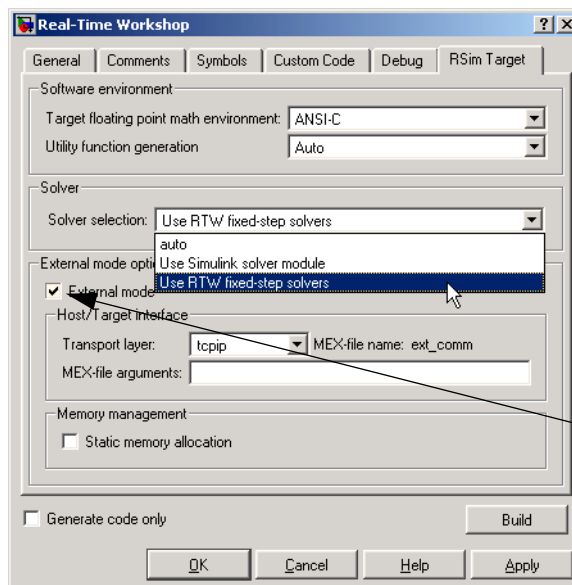
When you select the rapid simulation target, the name of the **Real-Time Workshop Interface** configuration dialog changes to **RSim Target**. In addition to the existing **Software environment** controls, the dialog contains solver and external mode options, as shown below:



The **Solver selection** option is set to auto by default. This causes a built-in solver to be used if a fixed-step solver is selected on the **Solver** pane, or will call the Simulink Solver module if a variable-step solver is chosen. You can instruct RSim to use a fixed-step solver by setting **Solver selection** to Use RTW fixed-step solvers. Note, however, that when you choose this option, you must have selected one of the fixed-step solvers in the **Solver** pane before you can build. If you want to use a variable-step solver, select one on the Solver pane, and set **Solver selection** in the Real-Time Workshop configuration dialog to Use Simulink solver module. When it runs, your executable attempts to check out a Simulink license in order to access the Solver module.

Note Rapid Simulation executables created without using the Simulink Solver module can be transferred and run on computers that do not have MATLAB installed. When an RSim executable runs on such a machine, it is necessary to have the following DLLs in your working directory: `libmx.dll`, `libut.dll`, and `libmat.dll`. These DLLs are required for the RSim executable to write and read data from a `.mat` file. This deployment option is not available for RSim executables that rely upon the Simulink Solver module.

If you want to run your executable in external mode, then select **External mode** in the External mode options section of the **RSim Target** dialog. The picture below shows the RSim-specific code generation and external mode options set to enable the RSim executable to communicate with Simulink:



Choose whether to generate code for a fixed-step or a variable-step solver with this pop-up menu. The auto option invokes the Simulink solver module only when the model requires it.

Select this check box to create an RSim executable that communicates with Simulink via external mode

Running a Rapid Simulation

The rapid simulation target lets you run a simulation similar to the Real-Time Workshop generic real-time (GRT) target. This simulation does not use timer

interrupts, and therefore is a nonreal-time simulation environment. The differences between GRT and RSim simulations are

- RSim supports variable-step solvers.
- RSim allows you to change parameter values or input signals at the start of a simulation without the need to generate code or recompile.

The GRT target, on the other hand, is a starting point for targeting a new processor.

A single build of your model can be used to study effects from varying parameters or input signals. Command-line arguments provide the necessary mechanism to specify new data for your simulation. The following table lists all available RSim command-line options.

RSim Command Line Options

Command-Line Option	Description
<code>model -f old.mat=new.mat</code>	Read From File block input signal data from a replacement MAT-file.
<code>model -L n</code>	Force the rapid simulation target to time out after n seconds.
<code>model -o newlogfile.mat</code>	Write MAT-file logging data to a file named newlogfile.mat.
<code>model -p filename.mat</code>	Read a new (replacement) parameter vector from a file named filename.mat.
<code>model -tf <stoptime></code>	Run the simulation until the time value <stoptime> is reached.
<code>model -t old.mat=new.mat</code>	The original model specified saving signals to the output file old.mat. For this run use the file new.mat for saving signal data.
<code>model -v</code>	Run in verbose mode.
<code>model -h</code>	Display a help message listing options.

Note On Solaris platforms, to run the RSim executable created for a model that uses variable-step solvers in a separate shell, the `LD_LIBRARY_PATH` environment variable is needed to indicate the path to the MATLAB installation directory, as follows:

```
% setenv LD_LIBRARY_PATH /apps/matlab/bin/sol2:$LD_LIBRARY_PATH
```

Enabling the Rapid Simulation Target to Time Out

Use the `-L timeout` execution option to enable the RSim executable to abort if it is taking excessive time. This can happen, for example, in some models when zero-crossings are frequent and minor step size is small.

To cause an executing RSim to time out after `n` seconds, use the `-L` command-line option followed by `n`. For example, suppose you created an RSim executable for the `vdp` demo; you can run the executable as follows:

```
vdp -L 20
```

After `vdp` (or `vdp.exe`) executes for 20 seconds the following happens:

On Windows platforms, the program will terminate with the message:

```
Exiting program, time limit exceeded
Logging available data ...
```

On UNIX platforms the message is

```
** 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 Real-Time Workshop configuration to use this feature. However, you must have generated the RSim executable using Version 6.0 or later of Real-Time Workshop for the `-L` flag to be recognized.

Obtaining the Parameter Structure from Your Model

To obtain a parameter structure for the current model settings you can use the `rsimgetrtp` function, with the following syntax:

```
rtP = rsimgetrtp('model', options)
```

The `rtP` structure is designed to be used with the Rapid Simulation target. Obtaining it via `rsimgetrtP` forces an *update diagram* action. In addition to the current model tunable block parameter settings, the `rtP` structure contains a structural checksum. This checksum is used to ensure that the model structure hasn't changed since the RSim executable was generated.

Options to `rsimgetrtP` are passed as parameter-value pairs. Currently there is one option, `AddTunableParamInfo`, which has two states, on and off:

```
rtP = rsimgetrtP('model','AddTunableParamInfo','on')
rtP = rsimgetrtP('model','AddTunableParamInfo','off')
```

The `AddTunableParamInfo` option causes Real-Time Workshop to generate code that extracts tunable parameter information from your model and places it in the return argument (`rtP`). This information gives you a mapping between the parameter structure and the tunable parameters.

To use the `AddTunableParamInfo` option, you must have selected the **Inline Parameters** check box in the **Optimization** pane of the **Configuration Parameters** dialog. Exercising this option also creates, then deletes, a `model.rtw` file in your current working directory.

Tunable fixed-point parameters are reported according to their 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`.

Example 1. Create an RSim executable and pass a different parameter structure:

- 1 Set the Real-Time Workshop target configuration to Rapid Simulation Target using the Target File Browser.
- 2 Create an RSim executable for the model by clicking the **Build** button or by typing `rtwbuild('model')`.
- 3 Modify parameters in your model and save the `rtP` structure:

```
rtP = rsimgetrtP('model')
save myrtP.mat rtP
```
- 4 Run the generated executable with the new parameter set:

```
!model -p myrtP.mat
```

5 Load the results into Matlab

```
load model.mat
```

Example 2. Create an rtP with the tunable parameter mapping information:

1 Create rtP with the tunable parameter information:

```
rtP = rsimgetrtP('model', 'AddTunableParamInfo', 'on')
```

2 The rtP structure contains

modelChecksum: 1-by-4 vector that encodes the structure of the model
parameters: structure of the tunable parameters in the model

3 The parameters structure contains the following member fields:

dataTypeName: the data type name, e.g., 'double'
dataTypeId: internal data type identifier for use by Real-Time Workshop
complex: 0 if real, 1 if complex

Specifying a New Signal Data File for a From File Block

To understand how to specify new signal data for a From File block, create a working directory and connect to that directory. Open the model `rsim_tfdemo` by typing

```
rsim_tfdemo
```

at the MATLAB prompt. Type

```
w = 100;  
zeta = 0.5;
```

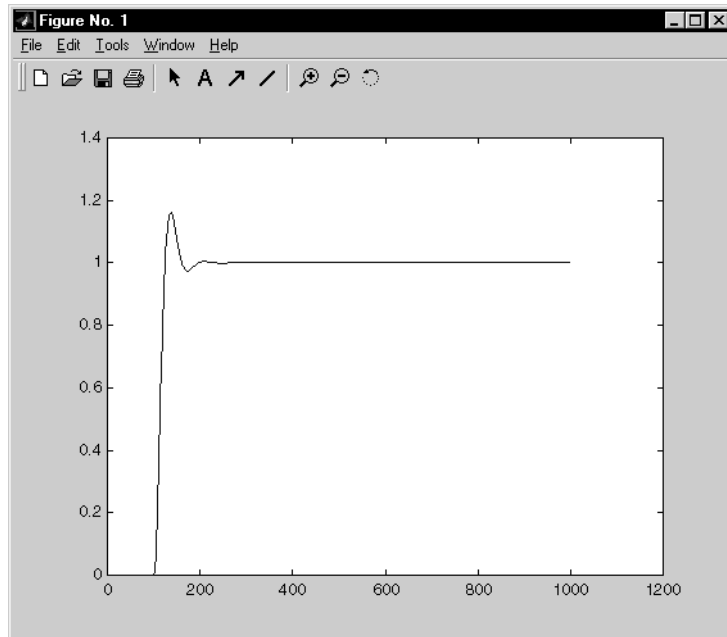
to set parameters. `rsim_tfdemo` requires a data file, `rsim_tfdemo.mat`. Make a local copy of `matlabroot/toolbox/rtw/rtwdemos/rsim_tfdemo.mat` in your working directory.

Be sure to specify `rsim.tlc` as the system target file and `rsim_default_tmf` as the template makefile. Then click the **Build** button on the **Real-Time Workshop** pane to create the RSim executable.

```
!rsim_tfdemo  
load rsim_tfdemo
```

```
plot(rt_yout)
```

The resulting plot shows simulation results using the default input data.



Replacing Input Signal Data. New data for a From File block can be placed in a standard MAT-file. As in Simulink, the From File block data must be stored in a matrix with the first row containing the time vector while subsequent rows contain u vectors as input signals. After generating and compiling your code, you can type the model name `rsim_tfdemo` at a system command prompt to run the simulation. In this case, the file `rsim_tfdat.mat` provides the input data for your simulation.

For the next simulation, create a new data file called `newfrom.mat` and use this to replace the original file (`rsim_tfdat.mat`), and run an RSim simulation with this new data. Type

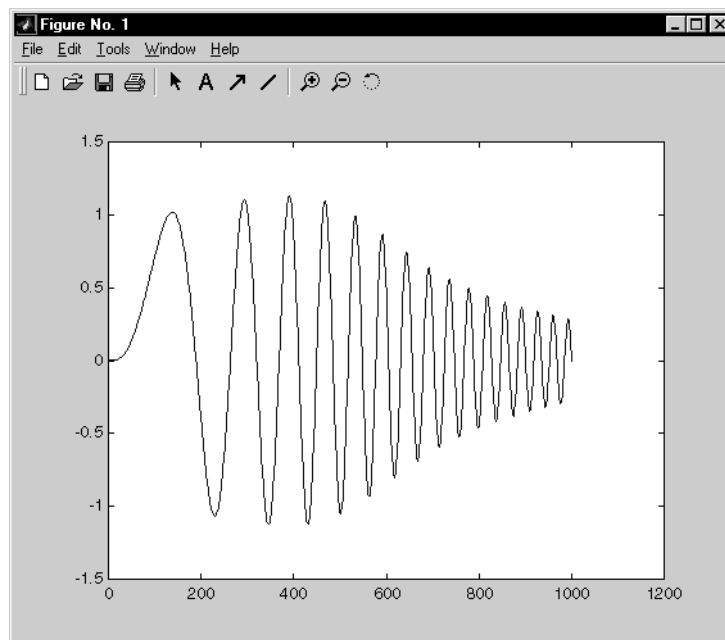
```
t=[0:.001:1];  
u=sin(100*t.*t);  
tu=[t;u];  
save newfrom.mat tu;
```

```
!rsimtfdemo -f rsim_tfddata.mat=newfrom.mat
```

at the MATLAB prompt. Now you can load the data and plot the new results by typing

```
load rsimtfdemo  
plot(rt_yout)
```

This picture shows the resulting plot.



As a result the new data file is read and the simulation progresses to the stop time specified in the **Solver** pane of the **Configuration Parameters** dialog box. It is possible to have multiple instances of From File blocks in your Simulink model.

Because RSim does not place signal data into generated code, it reduces code size and compile time for systems with large numbers of data points that originate in From File blocks. The From File block requires the time vector and signals to be data of type double. If you need to import signal data of a data type

other than double, use a From Workspace block with the data specified as a structure.

The workspace data must be in the format

```
variable.time  
variable.signals.values
```

If you have more than one signal, the format must be

```
variable.time  
variable.signals(1).values  
variable.signals(2).values
```

Specifying a New Output Filename for the Simulation

If you have specified any **Save to Workspace** options (that is, you selected the **Time, States, Outputs, or Final States** check boxes on the **Data Import/Export** pane of the **Configuration Parameters** dialog), the default is to save simulation logging results to the file *model.mat*. For example, the rapid simulation program *rsimtfdemo* normally saves data to *rsimtfdemo.mat*, as follows:

```
!rsimtfdemo  
created rsimtfdemo.mat
```

You can specify a new output filename for data logging by using the `-o` option when you run an executable:

```
!rsimtfdemo -o sim1.mat
```

In this case, the set of parameters provided at the time of code generation, including any From File block data, is run. You can combine a variety of RSim flags to provide new data, parameters, and output files to your simulation. See “Running a Rapid Simulation” on page 12-7 for a list of these flags.

Note that the MAT-file containing data for the From File blocks is required. This differs from the `grt` operation, which inserts MAT-file data directly into the generated C code that is then compiled and linked as an executable. In contrast, RSim allows you to provide new or replacement data sets for each successive simulation. A MAT-file containing From File or From Workspace data must be present if any From File or From Workspace blocks exist in your model.

Changing Block Parameters for an RSim Simulation

Once you have altered one or more parameters in the Simulink block diagram, you can extract the parameter vector, *model_P* (formerly *rtP*), for the entire model. The *model_P* vector, along with a model checksum, can then be saved to a MAT-file. This MAT-file can be read in directly by the stand-alone RSim executable, allowing you to replace the entire parameter vector quickly, for running studies of variations of parameter values where you are adjusting model parameters or coefficients or importing new data for use as input signals.

The model checksum provides a safety check to ensure that any parameter changes are only applied to RSim models that have the same model structure. If any block is deleted, or a new block added, then when a new *model_P* vector is generated, the new checksum will no longer match the original checksum. The RSim executable detects this incompatibility in parameter vectors and exits to avoid returning incorrect simulation results. In this case, where model structure has changed, you must regenerate the code for the model.

The RSim target allows you to alter any model parameter, including parameters that include *side-effects* functions. An example of a side-effects function is a simple Gain block that includes the following parameter entry in a dialog box.

```
gain value:  2 * a
```

In general, Real-Time Workshop evaluates side-effects functions before generating code. The generated code for this example retains only one memory location entry, and the dependence on parameter *a* is no longer visible in the generated code. The RSim target overcomes the problem of handling side-effects functions by replacing the entire parameter structure, *model_P*. You must create this new structure by using *rsimgetrtp.m* and then save it in a MAT-file. For the *rsimtfdemo* example, type

```
zeta = .2;  
myrtp = rsimgetrtp('modelname');  
save myparamfile myrtp;
```

at the MATLAB prompt.

In turn, RSim can read the MAT-file and replace the entire *model_P* structure whenever you need to 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 *myparamfile.mat*. In order to run the same *rsimtfdemo* model and use these new parameter values, execute the model by typing

```
!rsimtfdemo -p myparamfile.mat
load rsimtfdemo
plot(rt_yout)
```

Note that the *p* is lowercase and represents “parameter file.”

Specifying a New Stop Time for an RSim Simulation

If a new stop time is not provided, the simulation runs until it reaches the value specified in the **Solver** pane at the time of code generation. You can specify a new stop time value as follows:

```
!rsimtfdemo -tf 6.0
```

In this case, the simulation runs until it reaches 6.0 seconds. At this point it stops and logs the data using the MAT-file data logging rules as described above.

If your model includes From File blocks that also include a time vector in the first row of the time and signal matrix, the end of the simulation is still regulated by the original setting in the **Solver** pane of the **Configuration Parameters** dialog or from the *-s* option as described above. However, if the simulation time exceeds the endpoints of the time and signal matrix (that is, if the final time is greater than the final time value of the data matrix), then the signal data is extrapolated to the final time value as specified above.

Specifying New Output Filenames for To File Blocks

In much the same way as you can specify a new system output filename, you can also provide new output filenames for data saved from one or more To File blocks. This is done by specifying the original filename at the time of code generation with a new name as follows:

```
!mymodel -t original.mat=replacement.mat
```

In this case, assume that the original model wrote data to the output file called *original.mat*. Specifying a new filename forces RSim to write to the file *replacement.mat*. This technique allows you to avoid overwriting an existing simulation run.

Simulation Performance

It is not possible to predict accurately the simulation speedup of an RSim simulation compared to a standard Simulink simulation. Performance varies. Larger simulations have achieved speed improvements of up to 10 times faster than standard Simulink simulations. Some models might not show any noticeable improvement in simulation speed. The only way to determine speedup is to time your standard Simulink simulation and then compare its speed with the associated RSim simulation.

Batch and Monte Carlo Simulations

The RSim target is intended to be used for batch simulations in which parameters and/or input signals are varied for each new simulation. New output filenames allow you 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 under Microsoft Windows.

Create a file for Windows with any text editor and execute it by typing the filename, for example, mybatch, where the name of the text file is mybatch.bat.

```
rsimtfdemo -f rsimtfdemo.mat=run1.mat -o results1.mat -s 10.0
rsimtfdemo -f rsimtfdemo.mat=run2.mat -o results2.mat -s 10.0
rsimtfdemo -f rsimtfdemo.mat=run3.mat -o results3.mat -s 10.0
rsimtfdemo -f rsimtfdemo.mat=run4.mat -o results4.mat -s 10.0
```

In this case, batch simulations are run using the four sets of input data in files run1.mat, run2.mat, and so on. The RSim executable saves the data to the corresponding files specified after the -o option.

The variable names containing simulation results in each of these 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. If you want to avoid overwriting, you can copy the result to a new MATLAB variable before loading the next set of data.

You can also write M-file scripts to create new signals and new parameter structures, as well as to save data and perform batch runs using the bang command (!).

For additional insight into the rapid simulation target, explore rsimdemo1 and rsimdemo2, located in *matlabroot*/toolbox/rtw/rtwdemos/rsimdemos. These

examples demonstrate how RSim can be called repeatedly within an M-file for Monte Carlo simulations.

Limitations

The rapid simulation target is subject to the following limitations:

- The RSim target does not support algebraic loops
- The RSim target does not support MATLAB function blocks.
- The RSim target does not support noninlined M-file, FORTRAN and Ada S-functions.
- If an RSim build includes referenced models (via Model blocks), these models must be set up to use fixed-step solvers for code to be generated for them. The top model, however, can use a variable-step solver as long as all 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 would be changing the number of delays in a DSP simulation. In such cases, you must regenerate the code for the model.
- Variable-step solver support for RSim is not available on HP700, on IBM_RS platforms, or on PCWIN platforms using the following compiler versions:
 - Watcom C/C++ compiler version 10.6
 - Borland C/C++ compiler version 5.3

Targeting Tornado for Real-Time Applications

Tornado, a target supported by Real-Time Workshop, describes an integrated set of tools for creating real-time applications to run under the VxWorks operating system, which has many UNIX-like features and runs on a variety of host systems and target processors.

The Tornado Environment (p. 13-2)	Overview of the Tornado (VxWorks) Real-Time Target and the VxWorks block library
Run-Time Architecture Overview (p. 13-5)	Single-tasking and multitasking architecture and host/target communications
Implementation Overview (p. 13-12)	Design, implementation, and execution of a VxWorks-based real-time program using Real-Time Workshop

The Tornado Environment

This chapter describes how to create real-time programs for execution under VxWorks, which is part of the Tornado environment.

The VxWorks real-time operating system is available from Wind River Systems, Inc. It provides many UNIX-like features and comes bundled with a complete set of development tools.

Note Tornado is an integrated environment consisting of VxWorks (a high-performance real-time operating system), application building tools (compiler, linker, make, and archiver utilities), and interactive development tools (editor, debugger, configuration tool, command shell, and browser).

This chapter discusses the run-time architecture of VxWorks-based real-time programs generated by Real-Time Workshop and provides specific information on program implementation. Topics covered include

- Configuring device driver blocks and makefile templates
- Building the program
- Downloading the object file to the VxWorks target
- Executing the program on the VxWorks target
- Using the StethoScope data acquisition and graphical monitoring tool, which is available as an option with VxWorks. It allows you to access the output of any block in the model (in the real-time program) and display the data on the host.
- Using Simulink external mode to change model parameters and download them to the executing program on the VxWorks target. Note that you cannot use both external mode and StethoScope at the same time.

Confirming Your Tornado Setup Is Operational

Before beginning, you must install and configure Tornado on your host and target hardware, as discussed in the Tornado documentation. You should then run one of the VxWorks demonstration programs to ensure you can boot your VxWorks target and download object files to it. See the *Tornado User's Guide*

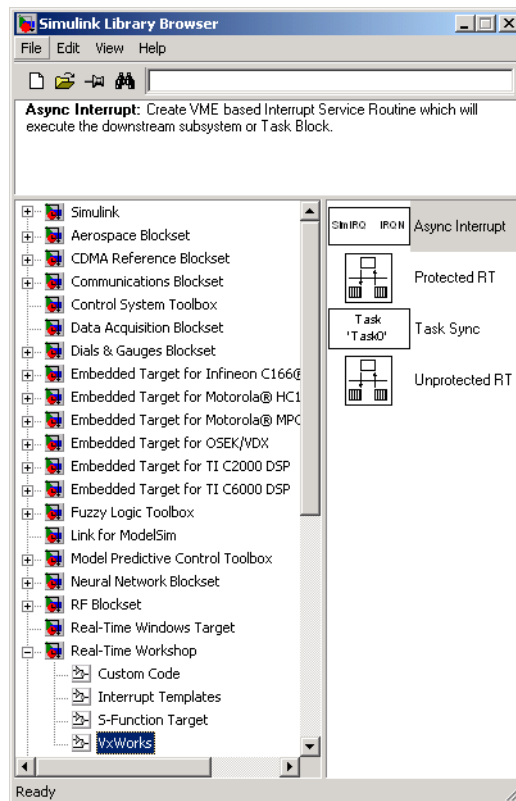
for additional information about installation and operation of VxWorks and Tornado products.

VxWorks Library

The VxWorks library (vxlib1) is part of the Real-Time Workshop library. You can access the VxWorks library by opening the Simulink Library Browser, clicking the plus sign to the left of the **Real-Time Workshop** entry, and clicking the **VxWorks** entry. Alternatively, type the following MATLAB command to open the VxWorks library directly:

```
vxlib1
```

The VxWorks library selected in the Simulink library browser is shown below.

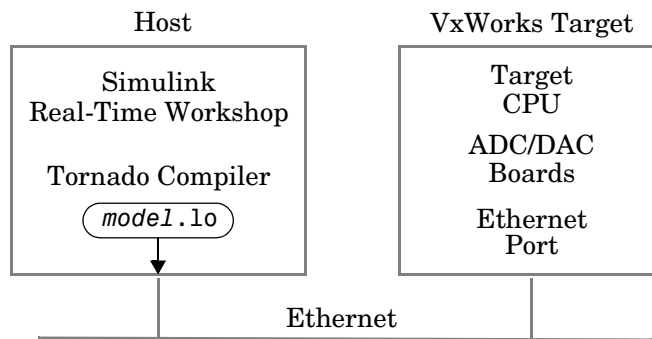


The VxWorks library blocks allow you to model and generate code for asynchronous event handling, including servicing of hardware-generated interrupts, maintenance of timers, asynchronous read and write operations, and spawning of asynchronous tasks under VxWorks. See Chapter 16, “Asynchronous Support,” for a detailed description of the blocks in the VxWorks library.

Note The older Interrupt Templates library (`vxlib`) is obsolete. It is provided only to allow models created prior to Real-Time Workshop 6.0 to continue to operate. If you have models that use `vxlib` blocks, we strongly recommend that you change them to use `vxlib1` blocks.

Run-Time Architecture Overview

In a typical VxWorks-based real-time system, the hardware consists of a UNIX or PC host running Simulink and Real-Time Workshop, connected to a VxWorks target CPU via Ethernet. In addition, the target chassis may contain I/O boards with A/D and D/A converters to communicate with external hardware. The following diagram shows the arrangement.



Typical Hardware Setup for a VxWorks Application

The real-time code is compiled on the UNIX or PC host using the cross compiler supplied with the VxWorks package. The object file (*model.io*) output from the Real-Time Workshop program builder is downloaded, using WindSh (the command shell) in Tornado, to the VxWorks target CPU via an Ethernet connection.

The real-time program executes on the VxWorks target and interfaces with external hardware via the I/O devices installed on the target.

Parameter Tuning and Monitoring

You can change program parameters from the host and monitor data with Scope blocks while the program executes, using Simulink external mode. You can also monitor program outputs using the StethoScope data analysis tool.

Using Simulink external mode or StethoScope allows you to change model parameters in your program, and to analyze the results of these changes, in real time.

External Mode

Simulink external mode provides a mechanism to download new parameter values to the executing program and to monitor signals in your model. In this mode, the external link MEX-file sends a vector of new parameter values to the real-time program via the network connection. These new parameter values are sent to the program whenever you make a parameter change, without requiring a new code generation or build iteration.

You can use the `rtBlockIOSignals` code generation option to monitor signals in VxWorks. See “C-API for Interfacing with Signals and Parameters” on page 17-2 for further information and example code.

The real-time program (executing on the VxWorks target) runs a low-priority task that communicates with the external link MEX-file and accepts the new parameters as they are passed into the program.

Communication between Simulink and the real-time program is accomplished using the sockets network API. This implementation requires an Ethernet network that supports TCP/IP. See Chapter 6, “External Mode,” for more information on external mode.

Changes to the block diagram structure (for example, adding or removing blocks) require regeneration of the model and execution of the build process.

Configuring VxWorks to Use Sockets

If you want to use Simulink external mode with your VxWorks program, you must configure your VxWorks kernel to support sockets by including the `INCLUDE_NET_INIT`, `INCLUDE_NET_SHOW`, and `INCLUDE_NETWORK` options in your VxWorks image. For more information on configuring your kernel, see the *VxWorks Programmer’s Guide*.

Before using external mode, you must ensure that VxWorks can properly respond to your host over the network. You can test this by using the host command

```
ping <target_name>
```

Note You might need to enter a routing table entry into VxWorks if your host is not on the same local network (subnet) as the VxWorks system. See `routeAdd` in the *VxWorks Reference Guide* for more information.

Configuring Simulink to Use Sockets

Simulink external mode uses a MEX-file to communicate with the VxWorks system. The MEX-file is

```
matlabroot/toolbox/rtw/rtw/ext_comm.*
```

where `*` is a host-dependent MEX-file extension. See Chapter 6, “External Mode,” for more information. Tornado only supports TCP/IP protocol.

To set up external mode with VxWorks,

- 1 Open Model Explorer or **Configuration Parameters** dialog
- 2 If you have not already done so, select the Tornado target using the **Browse** button on the **Real-Time Workshop/General** pane. For more information on how to configure a Tornado target, see “Specifying the Real-Time Build Options” on page 13-15.
- 3 Select the **Real-Time Workshop/Tornado Target** pane.
- 4 Select the **External mode** check box.

This causes two Host/Target interface options to appear below the check box. The first one, **Transport layer**, is preset to TCP/IP for Tornado targets and cannot be changed.

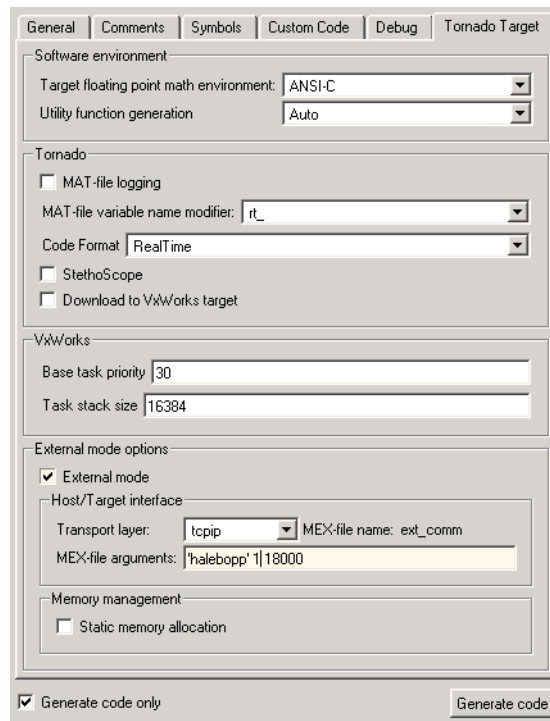
- 5 The second Host/Target interface option is the **MEX-file arguments** text field. In this field, type a list of arguments to be passed to the transport layer MEX-file. You must specify the name of the VxWorks target system and, optionally, the verbosity and TCP port number (see example below).

Verbosity can be 0 (the default) or 1 if extra information is desired. The TCP port number ranges from 256 to 65535 (the default is 17725).

If there is a conflict with other software using TCP port 17725, you can change the port that you use by editing the third argument of the **MEX-file arguments** field. The format for the MEX-file arguments is

```
'target_network_name' [verbosity] [TCP port number]
```

As an example, the **Tornado Target** pane of the **Real-Time Workshop Configuration Parameters** dialog below shows the external mode MEX-file arguments configured for a target system called halebopp with default verbosity of zero and the port assigned to 18000.



StethoScope

With StethoScope, you can access the output of any block in the model (in the real-time program) and display this data on a host. Signals are installed in StethoScope by the real-time program using the `rtBlockIOsignals` data structure (See “C-API for Interfacing with Signals and Parameters” on

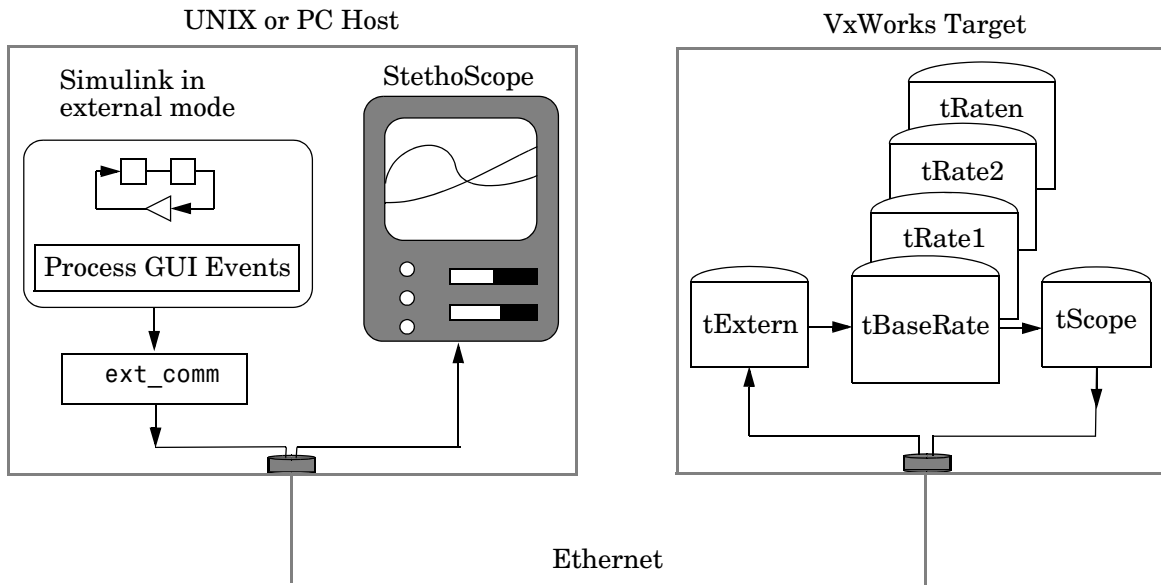
page 17-2 for information on `rtBlockIOsignals`), or interactively from the `WindSh` while the real-time program is running. To use `StethoScope` interactively, see the *StethoScope User's Manual*.

To use `StethoScope` you must specify certain options with the build command. See “Other Code Generation Options” on page 13-16 for information on these options.

Run-Time Structure

The real-time program executes on the VxWorks target while Simulink and `StethoScope` execute on the same or different host workstations. Simulink and `StethoScope` require tasks on the VxWorks target to handle communication.

This diagram illustrates the structure of a VxWorks application using Simulink external mode and `StethoScope`.



The Run-Time Structure

The program creates VxWorks tasks to run on the real-time system: one communicates with Simulink, the others execute the model. StethoScope creates its own tasks to collect data.

Host Processes

There are two processes running on the host side that communicate with the real-time program:

- Simulink running in external mode. Whenever you change a parameter in the block diagram, Simulink calls the external link MEX-file to download any new parameter values to the VxWorks target.
- The StethoScope user interface module. This program communicates with the StethoScope real-time module running on the VxWorks target to retrieve model data and plot time histories.

VxWorks Tasks

You can run the real-time program in either single-tasking or multitasking mode. The code for both modes is located in

```
matlabroot/rtw/c/tornado/rt_main.c
```

Real-Time Workshop compiles and links `rt_main.c` with the model code during the build process.

Single-tasking. By default, the model is run as one task, `tSingleRate`. This might actually provide the best performance (highest base sample rate) depending on the model.

The `tSingleRate` task runs at the base rate of the model and executes all necessary code for the slower sample rates. Execution of the `tSingleRate` task is normally blocked by a call to the VxWorks `semTake` routine. When a clock interrupt occurs, the interrupt service routine calls the `semGive` routine, which causes the `semTake` call to return. Once enabled, the `tSingleRate` task executes the model code for one time step. The loop then waits at the top by again calling `semTake`. For more information about the `semTake` and `semGive` routines, refer to the *VxWorks Reference Manual*. By default, it runs at a relatively high priority (30), which allows it to execute without interruption from background system activity.

Multitasking. Optionally, the model can run as multiple tasks, one for each sample rate in the model:

- `tBaseRate` — This task executes the components of the model code run at the base (highest) sample rate. By default, it runs at a relatively high priority (30), which allows it to execute without interruption from background system activity.
- `tRaten` — The program also spawns a separate task for each additional sample rate in the system. These additional tasks are named `tRate1`, `tRate2`, ..., `tRaten`, where `n` is the slowest sample rate in the system. The priority of each additional task is one lower than its predecessor (`tRate1` has a lower priority than `tBaseRate`).

Supporting Tasks. If you select external mode and/or StethoScope during the build process, these tasks are also created:

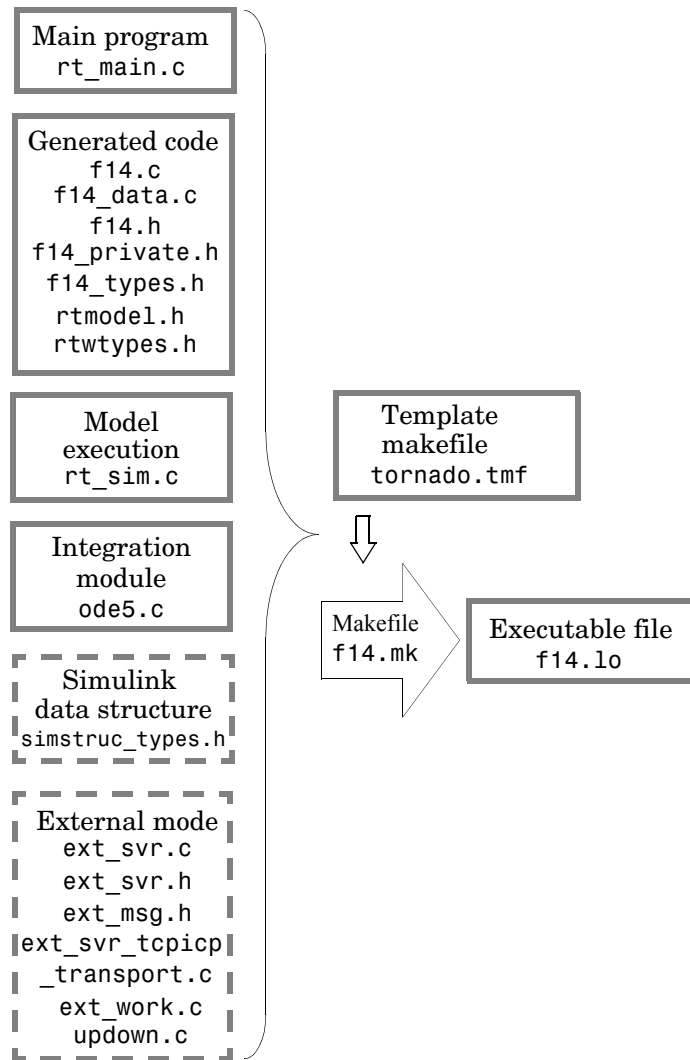
- `tExtern` — This task implements the server side of a socket stream connection that accepts data transferred from Simulink to the real-time program. In this implementation, `tExtern` waits for a message to arrive from Simulink. When a message arrives, `tExtern` retrieves it and modifies the specified parameters accordingly.
`tExtern` runs at a lower priority than `tRaten`, the lowest priority model task. The source code for `tExtern` is located in `matlabroot/rtw/c/src/ext_svr.c`.
- `tScopeDaemon` and `tScopeLink` — StethoScope provides its own VxWorks tasks to enable real-time data collection and display. In single-tasking mode, `tSingleRate` collects signals; in multitasking mode, `tBaseRate` collects them. Both perform the collection on every base time step. The StethoScope tasks then send the data to the host for display when there is idle time, that is, when the model is waiting for the next time step to occur. `rt_main.c` starts these tasks if they are not already running.

Implementation Overview

To implement and run a VxWorks-based real-time program using Real-Time Workshop, you must

- Design a Simulink model for your particular application.
- Add the appropriate device driver blocks to the Simulink model, if desired.
- Configure the `tornado.tmf` template makefile for your particular setup.
- Establish a connection between the host running Simulink and the VxWorks target via Ethernet.
- Use the automatic program builder to generate the code and the custom makefile, invoke the `make` command to compile and link the generated code, and load and activate the tasks required.

The figure below shows the Real-Time Workshop Tornado run-time interface modules and the generated code for the `f14` example model.



Source Modules Used to Build the VxWorks Real-Time Program

This diagram illustrates the code modules used to build a VxWorks real-time program. Dashed boxes indicate optional modules.

Adding Device Driver Blocks

The real-time program communicates with the I/O devices installed in the VxWorks target chassis via a set of device drivers. These device drivers contain the necessary code that runs on the target processor for interfacing to specific I/O devices.

To make device drivers easy to use, they are implemented as Simulink S-functions using C code MEX-files. This means you can connect them to your model like any other block and the code generator automatically includes a call to the block's C code in the generated code.

You can also inline S-functions via the Target Language Compiler. Inlining allows you to restrict function calls to only those that are necessary for the S-function. This can greatly increase the efficiency of the S-function. For more information about inlining S-functions, see Chapter 10, "Writing S-Functions for Real-Time Workshop," and the Target Language Compiler documentation.

You can have multiple instances of device driver blocks in your model.

Configuring the Template Makefile

To configure the VxWorks template, `tornado.tmf`, you must specify information about the environment in which you are using VxWorks. This section lists the lines in the file that you must edit.

Configuring `tornado.tmf` for VxWorks

To provide information used by VxWorks, you must specify the type of target and the specific CPU on the target. The target type is then used to locate the correct cross compiler and linker for your system.

The CPU type is used to define the CPU macro that is in turn used by many of the VxWorks header files. Refer to the *VxWorks Programmer's Guide* for information on the appropriate values to use.

This information is in the section labeled

```
#----- VxWorks Configuration -----
```

Edit the following lines to reflect your setup.

```
VX_TARGET_TYPE = 68k  
CPU_TYPE = MC68040
```


Downloading Configuration

In order to perform automatic downloading during the build process, the target name and host name that the Tornado target server are to run on must be specified. Modify these macros to reflect your setup.

```
#----- Macros for Downloading to Target-----  
TARGET = targetname  
TGTSVR_HOST = hostname
```

Tool Locations

In order to locate the Tornado tools used in the build process, the following three macros must either be defined in the environment or specified in the template makefile. Modify these macros to reflect your setup.

```
#----- Tool Locations -----  
WIND_BASE = c:/Tornado  
WIND_REGISTRY = $(COMPUTERNAME)  
WIND_HOST_TYPE = x86-win32
```

Building the Program

Once you have created the Simulink block diagram, added the device drivers, and configured the makefile template, you are ready to set the build options and initiate the build process.

Specifying the Real-Time Build Options

Because Tornado targets cannot access variable-step solvers from Simulink, you need to specify a fixed-step solver, as follows:

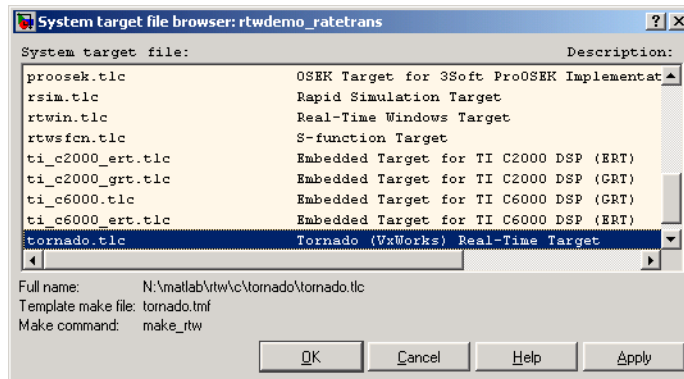
- 1 Open Model Explorer or **Configuration Parameters** dialog.
- 2 Select the **Solver** pane.
- 3 In the **Solver** pane, set the **Type** to Fixed-step.
- 4 Set the **Step Size** to the desired integration step size.
- 5 Select the integration algorithm; for models with continuous blocks, choose a fixed-step solver. If the model is purely discrete, set the integration algorithm to discrete (no continuous states).

6 Select the **Real-Time Workshop/General** pane.

7 Click the **Browse** button.

The System Target Browser opens.

8 Select Tornado (VxWorks) Real-Time Target, as shown below, and click **OK**.



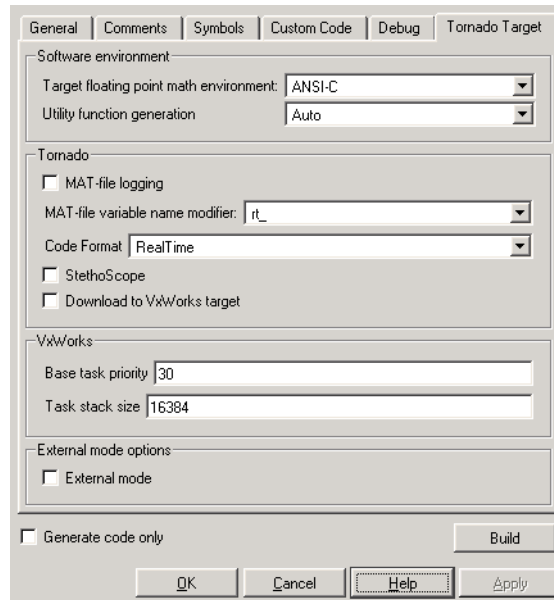
This sets the **Target configuration** options correctly, as shown below:

- **System target file:** tornado.tlc
- **TLC Options:** -p0 -aWarnNonSaturatedBlocks=0
- **Template makefile:** tornado.tmf template, which you must configure according to the instructions in “Configuring the Template Makefile” on page 13-14. (You can rename this file; simply change the dialog box accordingly.)
- **Make command:** make_rtw

You can optionally inline parameters for the blocks in the C code, which can improve performance. You can inline parameters when using external mode, while defining certain ones as tunable.

Other Code Generation Options. Once you have specified the Tornado target, as above, you can specify code generation options relevant to Tornado and VxWorks by selecting the **Tornado target** pane under the **Real-Time**

Workshop node in Model Explorer or the **Configuration Parameters** dialog. The default Tornado configuration is shown below:



Real-Time Workshop provides flags that set the appropriate macros in the template makefile, causing any necessary additional steps to be performed during the build process.

The flags and switches are as follows:

- **MAT-file logging:** Select this option to enable data logging during program execution. The program creates a file named *model.mat* at the end of program execution; this file contains the variables that you specified in the **Data Import/Export** pane of the **Configuration Parameters** dialog.

Real-Time Workshop adds a prefix or suffix to the names of the **Data Import/Export** pane variables that you select for logging. The **MAT-file variable name modifier** menu lets you select this prefix or suffix.

By default, the MAT-file is created in the root directory of the current default device in VxWorks. This is typically the host file system that VxWorks was booted from. Other remote file systems can be used as a destination for the

MAT-file using rsh or ftp network devices or NFS. See the *VxWorks Programmer's Guide* for more information. If a device or filename other than the default is desired, add " -DSAVEFILE=filename" to the OPTS flag to the make command. For example,

```
make_rtw OPTS=" -DSAVEFILE=filename"
```

- **Code format:** Selects RealTime or RealTimeMalloc code generation format.
- **StethoScope:** Select this option to enable the use of StethoScope with the generated executable. When starting rt_main, there are two command-line arguments that control the block names used by StethoScope; you can use them when starting the program on VxWorks. See the section “Running the Program” on page 13-21 for more information on these arguments.
- **Download to VxWorks target:** Enables automatic downloading of the generated program.
- **External mode:** Select this option to enable the use of external mode in the generated executable. You can optionally enable a verbose mode of external mode by appending -DVERBOSE to the OPTS flag in the make command. For example,

```
make_rtw OPTS=" -DVERBOSE"
```

causes parameter download information to be displayed on the console of the VxWorks system. For details on using external mode with Tornado, see “External Mode” on page 13-6.

Note External mode and StethoScope are mutually exclusive. If you enable **External mode**, you cannot enable the **StethoScope** option, and vice versa.

- **External mode transport:** tcp/ip is the only transport option for Tornado, and is selected by default.

Additional options are available on the **Real-Time Workshop** pane. See “Using the Real-Time Workshop Pane” on page 2-2 for information.

Initiating the Build

To build the program, click the **Build** button in the **Real-Time Workshop** pane of the **Configuration parameters** dialog. The resulting object file is named with the .lo extension (which stands for *loadable object*). This file is compiled

for the target processor using the cross compiler specified in the makefile. If automatic downloading (**Download to VxWorks target**) is enabled in the **Tornado code generation** options, the target server is started and the object file is downloaded and started on the target. If **StethoScope** is checked on the **Tornado code generation** options, you can now start StethoScope on the host. The StethoScope object files, `libxdr.so`, `libutilstssip.so`, and `libscope.so`, are loaded on the VxWorks target by the automatic download. See the *StethoScope User's Manual* for more information.

Resolving Header File Paths

Tornado 2.2.1 installs some standard header files in an include directory under the target compiler target directory. For example, if you are targeting the Motorola 68xxx processor for VxWorks with the GCC 2.96 compiler, Tornado installs the header files at the following location:

```
WIND_BASE/host/WIND_HOST_TYPE/lib/gcc-lib/m68k-wrs-vxworks/gcc-2
.96/include
```

To use Tornado 2.2.1 or higher with the Tornado (VxWorks) Real-Time Target, `tornado.tlc`, you must enable a macro in template makefile `tornado.tmf`. To enable the macro,

- 1 Open `matlabroot/rtw/c/tornado/tornado.tmf`.
- 2 Search for `TORNADO_TARGET_COMPILER_INCLUDES`.
- 3 Uncomment the macro `TORNADO_TARGET_COMPILER_INCLUDES` and set it to the include directory that contains the Tornado standard header files.

Given the path shown above, you would set the macro as follows:

```
TORNADO_TARGET_COMPILER_INCLUDES =
$(WIND_BASE)/host/$(WIND_HOST_TYPE)/lib/gcc-lib/m68k-wrs-vxworks
/gcc-2.96/include
```

Although this example shows the macro definition wrapped, you should include it on a single line.

If you are using a version of Tornado lower than 2.2.1, leave the macro commented out.

Downloading and Running the Executable Interactively

If automatic downloading is disabled, you must use the Tornado tools to complete the process. This involves three steps:

- 1 Establishing a communication link to transfer files between the host and the VxWorks target
- 2 Transferring the object file from the host to the VxWorks target
- 3 Running the program

Connecting to the VxWorks Target

After completing the build process, you are ready to connect the host workstation to the VxWorks target. The first step is ensuring that the Tornado registry (`wtxregd`) is running. Typically, it is configured to start running by the host operating system automatically at reboot. However, it can also be started interactively; see your Tornado documentation for details.

Next, start the target server. The target server provides communication between the Tornado tools on the host and the target agent on the target. You can do this either from a Windows command prompt or from within the Tornado development environment. From the Windows command prompt use

```
tgtsvr target_network_name -A -V
```

Additional options might be required, such as `-c` for the VxWorks core image location. Consult your Tornado documentation for details.

Downloading the Real-Time Program

To download the real-time program, use the VxWorks `ld` routine from within WindSh (wind shell). WindSh can also be run from the command line or from within the Tornado development environment.

For example, if you want to download the file `vx_equal.lo`, which is in the `/home/my_working_dir` directory, use the following commands at the WindSh prompt.

```
cd "/home/my_working_dir"  
ld <vx_equal.lo
```

You will also need to load the StethoScope libraries if the **StethoScope** option was selected during the build. The *Tornado User's Guide* describes the `ld` library routine.

Running the Program

The real-time program defines a function, `rt_main()`, that spawns the tasks to execute the model code and communicate with Simulink (if you selected external mode during the build procedure.) It also initializes StethoScope if you selected this option during the build procedure.

The `rt_main` function is defined in the `rt_main.c` application module. This module is located in the `matlabroot/rtw/c/tornado` directory.

The `rt_main` function takes six arguments, and is defined by the following ANSI C function prototype:

```
RT_MODEL * (*model_name)(void),
char_T     *optStr,
char_T     *scopeInstallString,
int_T      scopeFullNames,
int_T      priority,
int_T      port
```

The following table lists the arguments to this function.

Arguments to the `rt_main RT_MODEL`

Argument	Description
<code>model_name</code>	Pointer to the entry point function in the generated code. This function has the same name as the Simulink model. It registers the local functions that implement the model code by adding function pointers to the model's <code>rtM</code> . See Chapter 7, "Program Architecture," for more information.
<code>optStr</code>	<p>Options string used to specify a stop time (<code>-tf</code>) and whether to wait (<code>-w</code>) in external mode for a message from Simulink before starting the simulation. An example options string is</p> <pre>"-tf 20 -w"</pre> <p>The <code>-tf</code> option overrides the stop time that was set during code generation. If the value of the <code>-tf</code> option is <code>inf</code>, the program runs indefinitely.</p>
<code>scopeInstallString</code>	<p>Character string that determines which signals are installed to StethoScope. Possible values are</p> <ul style="list-style-type: none"> • <code>NULL</code> — Install no signals. This is the default value. • <code>"*"</code> — Install all signals. • <code>"[A-Z]*"</code> — Install signals from blocks whose names start with an uppercase letter. <p>Specifying any other string installs signals from blocks whose names start with that string.</p>
<code>scopeFullNames</code>	<p>Determines whether StethoScope uses full hierarchical block names for the signals it accesses or just the individual block name. Possible values are</p> <ul style="list-style-type: none"> • <code>1</code> Use full block names. • <code>0</code> Use individual block names. This is the default value. <p>It is important to use full block names if your program has multiple instances of a model or S-function.</p>

Arguments to the rt_main RT_MODEL (Continued)

Argument	Description
priority	Priority of the program's highest priority task (tBaseRate). Not specifying any value (or specifying a value of 0) assigns tBaseRate to the default priority, 30.
port	Port number that the external mode sockets connection should use. The valid range is 256 to 65535. The port number defaults to 17725.

Passing optStr via the Template Makefile. You can also pass the `-w` and `-tf` options (see `optStr` in the preceding table) to `rt_main` by using the `PROGRAM_OPTS` macro in `tornado.tmf`. `PROGRAM_OPTS` passes a string of the form

```
-opt1 val1 -opt2 val2
```

In the following examples, the `PROGRAM_OPTS` directive sets an infinite stop time and instructs the program to wait for a message from Simulink before starting the simulation. Note that the argument string must be delimited by single quotes nested within double quotes:

```
PROGRAM_OPTS = "'-tf inf -w'"
```

Including the single quotes ensures that the argument string is passed to the target program correctly under both Windows and UNIX.

Calling rt_main. To begin program execution, call `rt_main` from `WindSh`. For example,

```
sp(rt_main, vx_equal, "-tf 20 -w", "*", 0, 30, 17725)
```

- Begins execution of the `vx_equal` model
- Specifies a stop time of 20 seconds
- Provides access to all signals (block outputs) in the model by `StethoScope`
- Uses only individual block names for signal access (instead of the hierarchical name)
- Uses the default priority (30) for the `tBaseRate` task
- Uses TCP port 17725, the default

Custom Code Blocks

The following sections describe the Custom Code library, a collection of blocks that allow you to insert custom code into the generated source code files and functions associated with your model. This chapter includes the following topics:

Introduction (p. 14-2)

About the Custom Code library

Custom Code Library (p. 14-4)

Blocks that insert custom code into the generated model code

Introduction

The Custom Code library contains blocks that enable you to insert your own C code fragments into specific functions within code generated by Real-Time Workshop. These blocks are a superset of code customization capabilities built into the **Custom Code** configuration set dialog, and provide greater flexibility in terms of code placement than the built-in dialog.

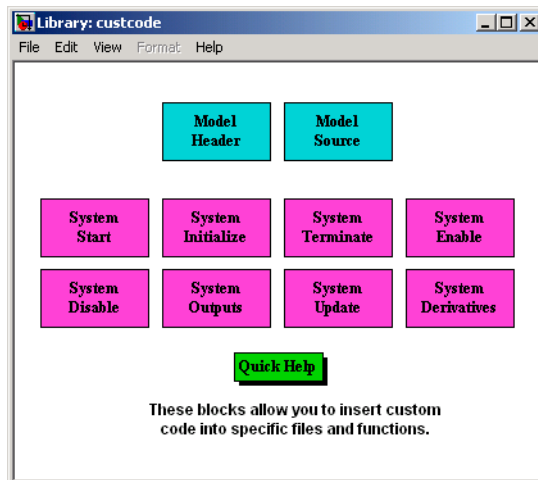
The Custom Code library is part of the Real-Time Workshop library. You can access the Real-Time Workshop library via the Simulink Library Browser. You can access Custom Code blocks via the Real-Time Workshop library by typing the MATLAB command

```
rtwlib
```

and then double-click the Custom Code Library block within it, or else type

```
custcode
```

to open the library directly. It is shown in the figure below.



This chapter discusses only the C Custom Code library.

The other sublibraries in the Real-Time Workshop library are

- **S-Function Target** — The S-Function Target sublibrary contains only one block type, the Real-Time Workshop S-Function block. This block is intended for use with generated S-functions. See Chapter 11, “The S-Function Target,” for more information.
- **VxWorks Library** — A collection of example blocks that support asynchronous interrupt servicing and task synchronization VxWorks. See Chapter 16, “Asynchronous Support,” for information. See also Chapter 13, “Targeting Tornado for Real-Time Applications,” for information on VxWorks.
- **Real-Time Windows Target** — Blocks used in building Real-Time Windows target applications (requires a Real-Time Windows license to use).

Custom Code Library

The Custom Code library contains blocks that allow you to place your own code and comments, in C, inside the code generated by Real-Time Workshop.

The library contains blocks that cause C code you place in them to be inserted in specific locations and functions in root models and subsystems. All Custom Code blocks except for Model Header and Model Source can be dragged into either root models or atomic subsystems. Model Header and Model Source blocks can only be placed in root models.

Note You can use models containing Custom Code blocks as submodels (models referenced by Model blocks). However, when simulation targets for submodels are generated, all Custom Code blocks within them are ignored. On the other hand, when submodel code is generated to create RTW targets, Custom Code is included and is compiled in the generated code.

The Custom Code library contains ten blocks that insert custom code into the generated model files and functions. You can view the blocks either by

- Expanding the C Custom Code node (under Real-Time Workshop library) in the Simulink Library Browser
- Right-clicking the Custom Code sublibrary icon in the right pane of the Simulink Library Browser

The latter method opens the window shown in the previous section.

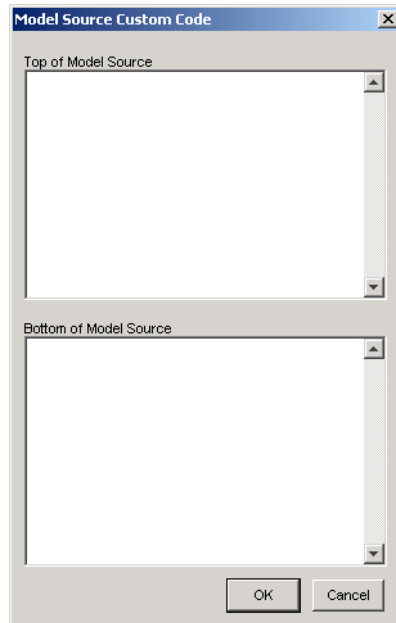
The two blocks on the top row contain text field entries where you can insert custom code at the top and bottom of

- *model.h* — Model Header File block
- *model.c* — Model Source File block

Each block contains two fields, in which you type or paste code and comments:

- Top of Model Source/Header
- Bottom of Model Source/Header

The figure below shows the Source Model block dialog:



The eight function blocks in the second and third rows contain text fields to insert custom code sections at the top and bottom of these designated model functions:

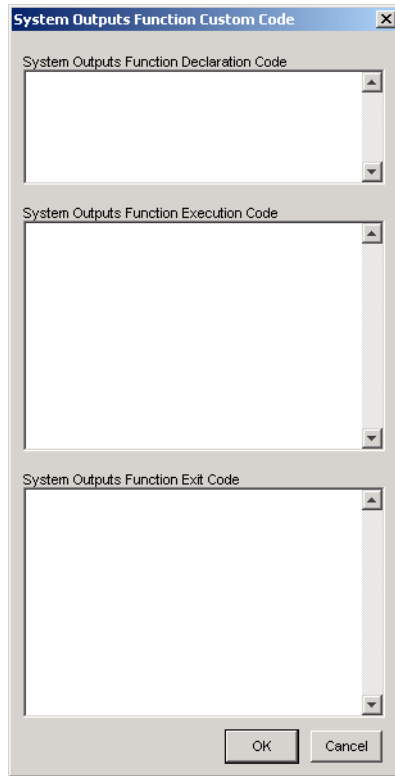
- `SystemStart` — System Start Function block
- `SystemInitialize` — System Initialize function block
- `SystemTerminate` — System Terminate function block
- `SystemEnable` — System Enable function block
- `SystemDisable` — System Disable function block
- `SystemOutputs` — System Outputs function block
- `SystemUpdate` — System Update function block
- `SystemDerivatives` — System Derivatives function block

Each of these blocks provides a dialog box that contains three fields:

- Declaration code

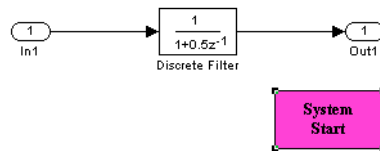
- Execution code
- Exit code

as shown in the **System Outputs Function Custom Code** dialog below:

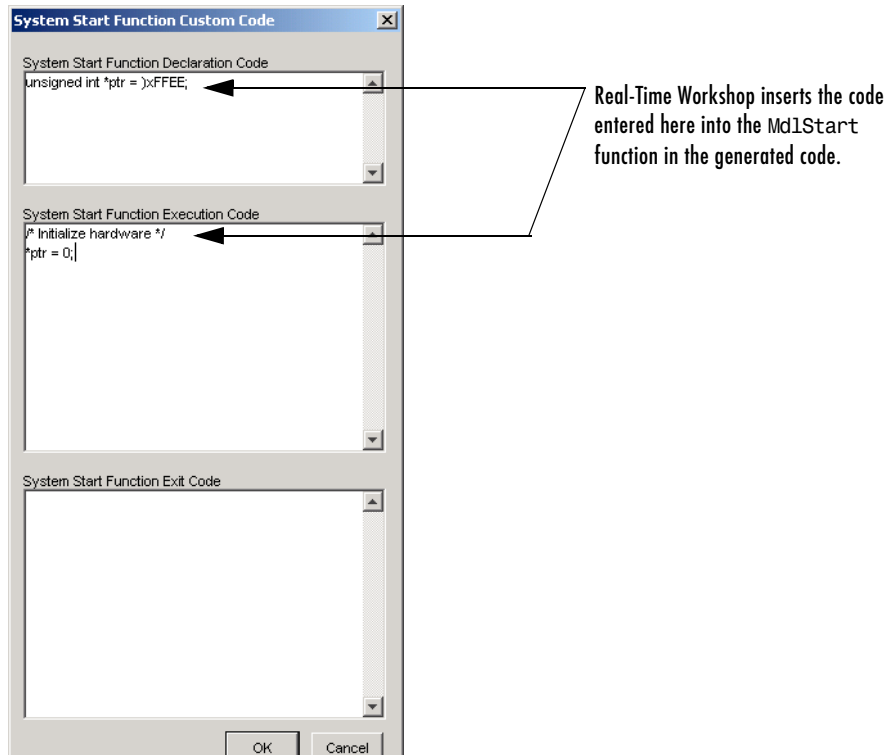


Example: Using a Custom Code Block

The following example uses a SystemStart function block to introduce code into the Md1Start function. The diagram below shows a simple model with the System Start function block inserted.



Double-clicking the System Start Function block opens the **System Start Function Custom Code** dialog box.



You can insert custom code into any or all of the available text fields. The code below is the Md1Start function for this example (mymodel).

```
void Md1Start(void)

{
  /* user code (Start function Header) */
  /* System: <Root> */
  unsigned int *ptr = 0xFFEE;

  /* user code (Start function Body) */
  /* System: <Root> */
  /* Initialize hardware */
  *ptr = 0;

  /* state initialization */
  /* DiscreteFilter Block: <Root>/Discrete Filter */
  rtX.d.Discrete_Filter = 0.0;
}
```

The custom code entered in the **System Start Function Custom Code** dialog box is embedded directly in the generated code. Note that each block of custom code is tagged with a comment such as

```
/* user code (Start function Header) */
```

Custom Code in Subsystems

The location of a Custom Code block in your model determines the location of the code it contains. You can use System Custom Code blocks either at root level or within atomic subsystems; the code is local to the subsystem in which you place the blocks. For example, the System Outputs block places code in `mdl0Outputs` when the code block resides in the root model. If the System Outputs block resides in a triggered or enabled subsystem, however, the code is placed in the subsystem's Outputs function.

The ordering for a triggered or enabled system is

- 1 Output entry
- 2 Output exit
- 3 Update entry
- 4 Update exit

Note If a root model or atomic subsystem does not need to generate a function for which a Custom Code block has been supplied, either the code in the block is not used or an error is generated. There is no diagnostic setting to control this. To eliminate the error, remove the Custom Code block.

Timing Services

Absolute and Elapsed Time
Computation (p. 15-2)

Describes efficient, overflow-free integer time computation services provided to blocks that request absolute or elapsed time

C and TLC APIs for Accessing Timers
(p. 15-5)

APIs that let you access timers for use in your S-functions, in both simulation and code generation

Elapsed Timer Code Generation
Example (p. 15-9)

Analysis of how a simple model generates and maintains an elapsed timer

Absolute and Elapsed Time Computation

Certain blocks require the value of either *absolute* time (i.e., the time from the start of program execution to the present time) or *elapsed* time (for example, the time elapsed between two trigger events). All 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, on an as-needed basis. If no blocks executing at a given rate require a timer, no timer is 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.
- Real-Time Workshop provides S-function and TLC APIs that let your S-functions access timers, in both simulation and code generation.
- For ERT and ERT-derived targets, the word size of the timers is determined by a user-specified maximum counter value. Correct specification of this value ensures that timers will not overflow. See the description of the parameter “Application Lifespan” on page 2-22. See also the Real-Time Workshop Embedded Coder documentation for information on restrictions on its use.

See “Blocks That Depend on Absolute Time” on page A-1 for a complete list of blocks in this category.

Timers for Periodic and Asynchronous Tasks

This chapter discusses timing services provided for blocks executing within *periodic* tasks (i.e., tasks running at the model’s base rate or subrates).

The Real-Time Workshop also provides timer support for blocks whose execution is *asynchronous* with respect to the periodic timing source of the model. See the following sections of the Asynchronous Support chapter:

- “Using Timers in Asynchronous Tasks” on page 16-29
- “Creating a Customized Asynchronous Library” on page 16-31

Allocation of Timers

If a block requires absolute or elapsed time data, it must register the requirement (see “C and TLC APIs for Accessing Timers” 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.
- One block running at rate A registers a requirement for absolute time.
- No blocks running at rate B require absolute or elapsed time.
- Two blocks running at rate C register 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. In the current release, a default size of 64 bits is used. This is the amount of memory allocated for a timer if you specify a value of `inf` for the **Application lifespan (days)** parameter. For an application whose sample rate is 1000 mHz, a 64-bit counter will not overflow for more than 500 years.

For information on how you can control the the word size used for timers, see the description of the **Application lifespan (days)** parameter in “Application Lifespan” on page 2-22.

Elapsed Time Counters in Triggered Subsystems

Some blocks (such as the Discrete-Time Integrator block), perform computations requiring the elapsed time (ΔT) since the previous block execution. These blocks present a special problem when executing within a triggered subsystem.

In previous releases, such blocks maintained a previous-time value and executed an elapsed time computation each time they ran. If a triggered subsystem contained multiple blocks requiring elapsed time data, the same elapsed time computation would be performed redundantly by all such blocks.

In this release, generation of redundant timers has been eliminated. Blocks requiring elapsed time data must register the requirement (see “C and TLC APIs for Accessing Timers” 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 any unconditionally executed subsystem within the triggered subsystem) contains one or more blocks requiring elapsed time data.

C and TLC APIs for Accessing Timers

This section describes APIs that let your S-functions take advantage of the efficiencies offered by the 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`.
- Existing S-functions that read absolute time but do not register via these macros will continue to operate correctly, but will generate old-style, less efficient code.

C API for S-Functions

The SimStruct macros described in this section provide access to absolute and elapsed timers for S-functions during simulation. All the functions described (except `ssGetTaskTime`) are new in Release 14.

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 Writing S-functions 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 15-3.

- `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 (i.e., 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 greater than 32 bits, the value is returned as an array of two 32-bit words, with the low-order word stored at the lower address.

Note that the integer counter value is *not* equivalent to the actual elapsed time; the actual elapsed time is obtained by computing

```
realElapseTime = counter_value * resolution;
```

where `counter_value` is the integer value returned by `ssGetElapseTimeCounter`, and `resolution` is the resolution of the counter returned by `ssGetElapseResolution`.

To determine how to access the returned counter value, obtain the data type of the counter by calling `ssGetElapseTimeCounterDtype`, as in the following code fragment:

```

ssGetElapseTime( S, y_elapseTime);
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 elapse time counter");
break;
}
}

```

TLC API for Code Generation

The following TLC functions are provided to support elapsed time counters in generated code. All the functions described (except `LibGetTaskTimeFromTID`) are new in Release 14.

- `LibGetTaskTimeFromTID(block)`: Generates code to read the absolute time for the task in which `block` executes.

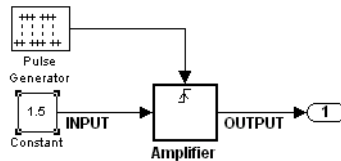
`LibGetTaskTimeFromTID` is documented with other sample time functions in the TLC Function Library Reference pages of the Target Language Compiler documentation.

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.

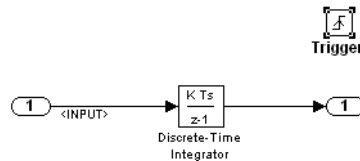
- `LibGetElapseTime(system)`: Generates code to read the elapsed time counter for `system`. (`system` is the parent system of the calling block.) See “Elapsed Timer Code Generation Example” on page 15-9 for an illustration of code generated by this function.
- `LibGetElapseTimeCounter(system)`: Generates 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.)

Elapsed Timer Code Generation Example

This section shows a simple model illustrating how an elapsed time counter is generated and used by a Discrete-Time Integrator block within a triggered subsystem. The following block diagrams show the model (`elapsedTime_exp`), which contains subsystem (`Amplifier`) that includes a Discrete-Time Integrator block.



`elapsedTime_exp` Model



Amplifier Subsystem

A 32-bit timer for the base rate (the only rate in this model) is defined within the `rtModel` structure, as follows, in `model.h`.

```

/*
 * Timing:
 * The following substructure contains information regarding
 * the timing information for the model.
 */
struct {
    time_T stepSize;
    uint32_T clockTick0;
    uint32_T clockTickH0;
    time_T stepSize0;

```

```
time_T tStart;
time_T tFinal;
time_T timeOfLastOutput;
void *timingData;
real_T *varNextHitTimesList;
SimTimeStep simTimeStep;
boolean_T stopRequestedFlag;
time_T *sampleTimes;
time_T *offsetTimes;
int_T *sampleTimeTaskIDPtr;
int_T *sampleHits;
int_T *perTaskSampleHits;
time_T *t;
time_T sampleTimesArray[1];
time_T offsetTimesArray[1];
int_T sampleTimeTaskIDArray[1];
int_T sampleHitArray[1];
int_T perTaskSampleHitsArray[1];
time_T tArray[1];
} Timing;
```

Note that had the target been ERT instead of GRT, the Timing struct would have been pruned to contain only the data required by the model, as follows:

```
/* Real-time Model Data Structure */ (for ERT!)
struct _RT_MODEL_elapseTime_exp_Tag {

    /*
     * Timing:
     * The following substructure contains information regarding
     * the timing information for the model.
     */
    struct {
        uint32_T clockTick0;
    } Timing;
};
```

Storage for the previous-time value of the Amplifier subsystem (Amplifier_PREV_T) is allocated in the D_Work (states) struct in `model.h`.

```
typedef struct D_Work_elapseTime_exp_tag {
    real_T DiscreteTimeIntegrator_DSTATE; /* '<S1>/Discrete-Time Integrator' */
    int32_T clockTickCounter;             /* '<Root>/Pulse Generator' */
    uint32_T Amplifier_PREV_T;            /* '<Root>/Amplifier' */
} D_Work_elapseTime_exp;
```

These structures are declared in `model.c`:

```
/* Block states (auto storage) */
D_Work_elapseTime_exp elapseTime_exp_DWork;
.
.
.
/* Real-time model */
rtModel_elapseTime_exp elapseTime_exp_M;
rtModel_elapseTime_exp *elapseTime_exp_M = &elapseTime_exp_M;
```

The elapsed time computation is performed as follows within the `model_step` function:

```
/* Output and update for trigger system: '<Root>/Amplifier' */
uint32_T rt_currentTime =
    ((uint32_T)elapseTime_exp_M->Timing.clockTick0);
uint32_T rt_elapseTime = rt_currentTime -
    elapseTime_exp_DWork.Amplifier_PREV_T;
elapseTime_exp_DWork.Amplifier_PREV_T = rt_currentTime;
```

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 = elapseTime_exp_DWork.DiscreteTimeIntegrator_DSTATE;

/* Update for DiscreteIntegrator: '<S1>/Discrete-Time Integrator' */
elapseTime_exp_DWork.DiscreteTimeIntegrator_DSTATE += 0.3 *
    (real_T)rt_elapseTime * 1.5;
```

Note that because the triggered subsystem maintains the elapsed time, the TLC implementation of the Discrete-Time Integrator block needs only a single call to `LibGetElapsedTime` to access the elapsed time value.

Asynchronous Support

Introduction (p. 16-2)

Overview of the asynchronous blocks in the Real-Time Workshop libraries

Interrupt Handling Blocks (p. 16-5)

Detailed description of the Async Interrupt, Task Synchronization, and Protected/Unprotected Rate Transition blocks

Rate Transitions and Asynchronous Blocks (p. 16-23)

How and when to use Rate Transition blocks to handle data transfers to and from asynchronous blocks

Using Timers in Asynchronous Tasks (p. 16-29)

How absolute and elapsed timing data is maintained for use of blocks executing in an asynchronous task

Creating a Customized Asynchronous Library (p. 16-31)

Guidelines for creating your own asynchronous blocks, using the VxWorks library blocks as templates; required API calls for C and TLC block implementations

Introduction

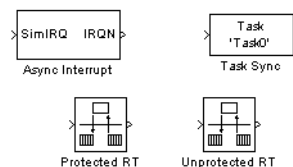
Real-Time Workshop models are normally timed from a *periodic* interrupt source (e.g., a hardware timer). Blocks in a periodically clocked single-rate model run at a timer interrupt rate (the base rate of the model). Blocks in a periodically clocked multirate model run at the base rate or at submultiples 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 appropriately, for example, by acquiring data from the interrupting device.

This chapter describes a library of blocks that allow you 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). Although the blocks described target a particular RTOS (VxWorks Tornado), this chapter also provides source code analysis and other information that enables you to develop blocks supporting asynchronous event handling for your target RTOS.

VxWorks Library Overview

The following figure shows the blocks in the VxWorks library (vxlib1).



The key blocks in the library are the Async Interrupt and Task Synchronization blocks. These blocks are targeted for the VxWorks Tornado operating system. You can use them, without modification, to support VxWorks applications.

If you want to implement asynchronous support for an RTOS other than VxWorks, guidelines and example code are provided that will help you to adapt the VxWorks library blocks to target your RTOS. This topic is discussed in “Creating a Customized Asynchronous Library” on page 16-31.

The VxWorks library includes the following VxWorks-specific blocks:

- **Async Interrupt block:** Generates interrupt-level code. Each output of the Async Interrupt block is associated with a user-specified VxWorks VME interrupt. When an output is connected to the control input of a triggered subsystem such as a function call subsystem, the generated subsystem code is called from an interrupt service routine (ISR).
- **Task Synchronization block:** Function Call subsystem that spawns an independent VxWorks task that calls the function call subsystem connected to its output. The Task Synchronization block is designed to work in conjunction with the Async Interrupt block connected to its control input.

The VxWorks library also includes blocks that are not target specific. These blocks support data transfers between blocks running at different priorities:

- **Protected Rate Transition block:** The Rate Transition block that is configured to ensure data integrity during data transfers between blocks running as different tasks
- **Unprotected Rate Transition block:** The Rate Transition block that is configured to operate in unprotected / nondeterministic mode during data transfers between blocks running as different tasks

The Protected and Unprotected Rate Transition blocks are provided as a convenience. You can use the built-in Simulink Rate Transition block for the same purpose. The use of the Protected and Unprotected Rate Transition blocks in asynchronous contexts is discussed in “Rate Transitions and Asynchronous Blocks” on page 16-23. For general information on rate transitions, see Chapter 8, “Models with Multiple Sample Rates.”

Accessing the VxWorks Library

The VxWorks library (vxlib1) is part of the Real-Time Workshop library. You can access the VxWorks library by opening the Simulink Library Browser, clicking the plus sign to the left of the **Real-Time Workshop** entry, and clicking the **VxWorks** entry.

Alternatively, type the following MATLAB command to open the VxWorks library directly:

```
vxlib1
```

Note The older Interrupt Templates library (`vxlib`) is obsolete. It is provided only to allow models created prior to Real-Time Workshop 6.0 to continue to operate. If you have models that use `vxlib` blocks, we strongly recommend that you change them to use `vxlib1` blocks.

Generating Code with the VxWorks Library Blocks

To generate a VxWorks compatible application from a model containing VxWorks library blocks, configure the model for one of the following targets:

- The Embedded Real-Time (ERT) target. This target is provided with the Real-Time Workshop Embedded Coder.
When using the ERT target with VxWorks library blocks, you must select the **Generate an example main program** option, and select `VxWorksExample` from the **Target operating system** pop-up menu.
- The Tornado (VxWorks) Real-Time target example, included with Real-Time Workshop (see Chapter 13, “Targeting Tornado for Real-Time Applications”).

Demos and Additional Information

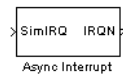
Additional information relevant to the topics in this chapter can be found in

- The `asyncdemo` model. To open this demo, type `asyncdemo` at the MATLAB command prompt.
- Chapter 8, “Models with Multiple Sample Rates,” discusses general multitasking and rate transition issues for periodic models.
- Chapter 13, “Targeting Tornado for Real-Time Applications,” discusses the Tornado (VxWorks) Real-Time target example.
- The Real-Time Workshop Embedded Coder documentation discusses the Embedded Real-Time (ERT) target, including task execution and scheduling.
- See your VxWorks system documentation for detailed information about the VxWorks system calls mentioned in this chapter.

Interrupt Handling Blocks

This section describes the VxWorks library blocks and their parameters in detail.

Async Interrupt Block



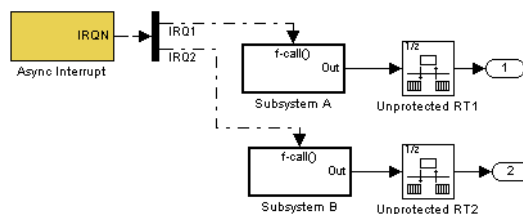
The primary purpose of the Async Interrupt block is to generate interrupt service routines (ISRs) associated with a specific VxWorks VME interrupt level. The Async Interrupt block enables the specified interrupt level and installs an ISR that calls the connected function call subsystem.

The Async Interrupt block can also be used in a simulation. It provides an input port that can be enabled and connected to a simulated interrupt source.

Connecting the Block. To generate an ISR, connect an output of the VxWorks Async Interrupt block to the control input of

- A function call subsystem
- The input of a VxWorks Task Synchronization block
- The input to a Stateflow chart configured for a function call input event

The figure below 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:

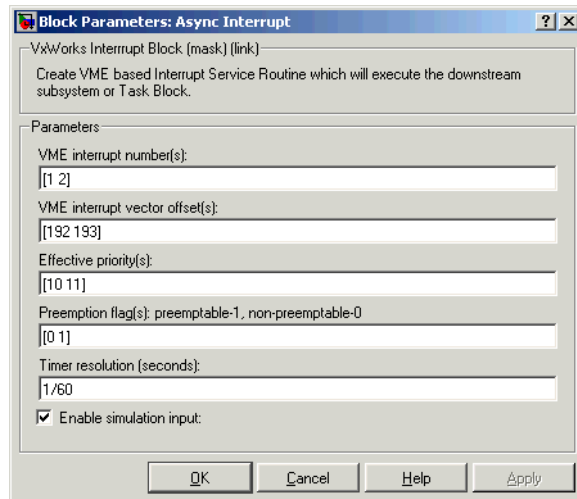
- The Async Interrupt block supports VME interrupts 1 through 7.
- The Async Interrupt block requires a VxWorks Board Support Package (BSP) that supports the following VxWorks system calls:
 - `sysIntEnable`
 - `sysIntDisable`
 - `intConnect`
 - `intLock`
 - `intUnlock`
 - `tickGet`

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 Synchronization block to synchronize the execution of the function call subsystem to a VxWorks task. The Task Synchronization block is placed between the Async Interrupt block and the function call subsystem. The Async Interrupt block then installs the Task Synchronization 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 VxWorks. See “Task Synchronization Block” on page 16-15 for more information.

Async Interrupt Block Parameters

This figure shows the Async Interrupt block dialog box.



The Async Interrupt block parameters are

- **VME interrupt number(s):** Specify an array of VME interrupt numbers for the interrupts to be installed. The valid range is 1 . . 7.
The width of the Async Interrupt block output signal corresponds to the number of VME interrupt numbers specified.

Note There can be more than one Async Interrupt block in a model. However, if you use more than one Async Interrupt block, there must be no duplication of the VME interrupt numbers specified in each block.

- **VME interrupt vector offset(s):** Specify an array of unique interrupt vector offset numbers corresponding to the VME interrupt numbers entered in the **VME interrupt number(s)** field. Real-Time Workshop passes the offsets to the VxWorks call `intConnect(INUM_TO_IVEC(offset), . . .)`.
- **Effective priority:** Each output of the Async Interrupt block drives a downstream block (e.g., a function call subsystem). The **Effective priority** specifies the Simulink priority of the downstream blocks. Specify an array of

priorities corresponding to the VME interrupt numbers entered in the **VME interrupt number(s)** field.

The **Effective priority** values are required to generate the proper rate transition code (see “Rate Transitions and Asynchronous Blocks” on page 16-23). Effective priority values are also required to ensure absolute time integrity when the asynchronous task needs to obtain real time from its base rate or its caller. The assigned priorities typically are higher than the priorities assigned to periodic tasks.

- **Preemption Flag(s)**: By default, higher priority interrupts can preempt lower priority interrupts in VxWorks. However, you can lock out interrupts during the execution of an ISR by setting the preemption flag to 0. This causes generation of `intLock()` and `intUnlock()` calls at the beginning and end of the ISR code. Interrupt locking should be used carefully, as it increases the system’s interrupt response time for all interrupts at the `intLockLevelSet()` level and below. Specify an array of flags corresponding to the VME interrupt numbers entered in the **VME interrupt number(s)** field.

Note that when an output of the Async Interrupt block drives a Task Synchronization block, the corresponding preemption flag must be set to 1.

Note The number of elements in the arrays specifying **VME interrupt vector offset(s)** and **Effective priority** must match the number of elements in the **VME interrupt number(s)** array.

- **Timer resolution (seconds)**: ISRs generated by the Async Interrupt block maintain their own absolute time counters. By default, these timers obtain their values from the VxWorks kernel via the `tickGet` call. The **Timer resolution** field determines the resolution of these counters. The default resolution is 1/60 second. The `tickGet` resolution for your BSP might be different. You should determine the `tickGet` resolution for your BSP and enter it in the **Timer resolution** field.

Note that if you are targeting VxWorks, you can obtain better timer resolution by replacing the `tickGet` call and accessing a hardware timer via your BSP instead. If you are targeting an RTOS other than VxWorks, you should replace the `tickGet` call with an equivalent call to the target RTOS, or generate code to read the appropriate timer register on the target

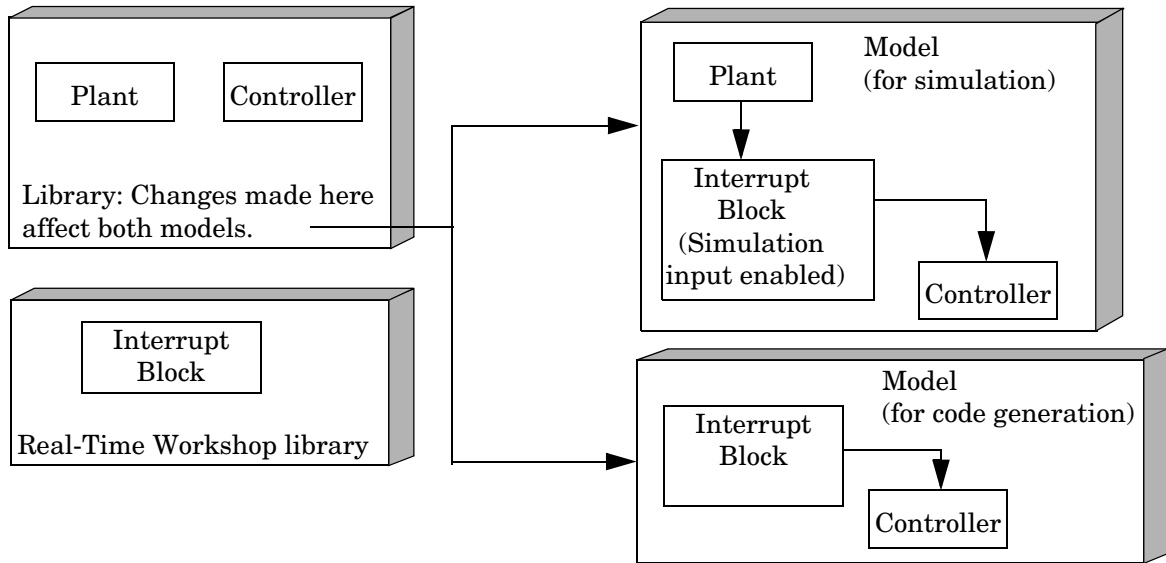
hardware. See “Using Timers in Asynchronous Tasks” on page 16-29 and “Async Interrupt Block Implementation” on page 16-31 for further information.

By default, timer values are stored as 32-bit integers. However, you can indirectly control the word size of the counters by setting the **Application lifespan (days)** option. See “Application Lifespan” on page 2-22. See also “Using Timers in Asynchronous Tasks” on page 16-29. An overflow warning is issued if the lifespan is too large to be handled by a 32-bit integer at the given time resolution.

- **Enable simulation input:** When this option is selected, an input port is added to the Async Interrupt block. This port is for use in simulation only. Connect one or more simulated interrupt sources to the simulation input. See the following section for further 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 following figure illustrates how changes made to the plant or controller, both of which are in a library, are propagated to the models.



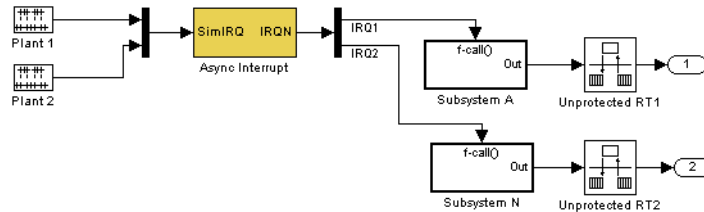
Dual-Model Use of Async Interrupt Block for Simulation and Code Generation

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 `asyncdemo` model.

Dual-Model Approach: Simulation

The following two block diagrams below illustrate simulation and code generation versions of a simple mode, respectively.

Simulated Interrupt Signals



Dual-Model Approach: Simulation Version of Model

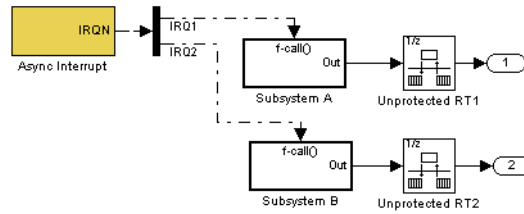
In the simulation version of the model, the simulation input of the Async Interrupt block is enabled. The input receives simulated interrupt signals from Plant1 and Plant2. Simulated interrupt signals are routed through the Async Interrupt block's input port. Upon receiving a simulated interrupt, the block calls the connected subsystem.

During simulation, subsystems connected to Async Interrupt block outputs are executed in order of their VxWorks priority. 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 VxWorks priorities. (In effect, in this case the Async Interrupt block behaves as if all inputs received a 1 simultaneously.)

Dual-Model Approach: Code Generation

In the code generation version of the model, the simulation port is disabled and the simulated plant signals that were included in the previous example have been removed.



Stand-alone functions are installed as ISRs.

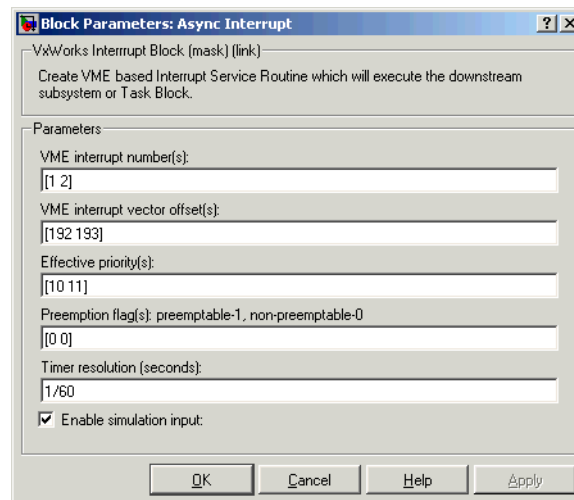
Interrupt Vector Table

Offset

192	&isr_num1_vec192()
193	&isr_num2_vec193()

Dual-Model Approach: Code Generation Version of Model

Consider the code generated from this model, assuming that the Async Interrupt Block parameters are configured as shown in this 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 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);
```

Note that 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 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 by the VxWorks `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 substrate 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 “Using Timers in Asynchronous Tasks” on page 16-29 for details.

Model Termination Code. The model's termination function disables the interrupts:

```

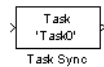
/* Model terminate function */
void Async_Code_terminate(void)
{

    /* 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);
}

```

Task Synchronization Block



The VxWorks Task Synchronization block is a function call subsystem that spawns an independent VxWorks task. The task calls the function call subsystem connected to the output of the Task Synchronization block.

Typically the Task Synchronization block is placed between an Async Interrupt block and a function call subsystem block or a Stateflow chart. Another example would be to place the Task Synchronization block at the output of a Stateflow diagram that has an event, “Output to Simulink,” configured as a function call.

The VxWorks Task Synchronization block performs the following functions:

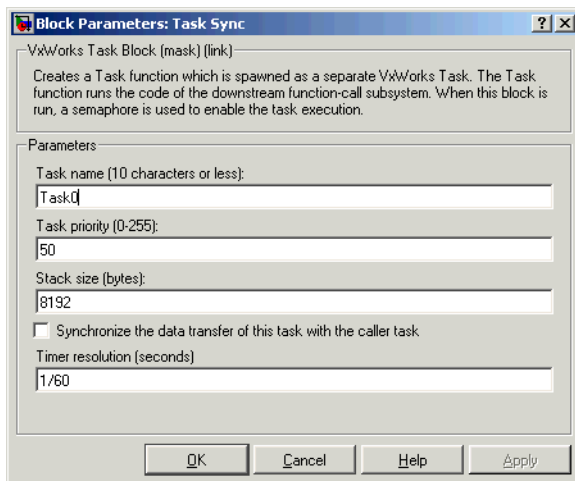
- An independent task is spawned, using the VxWorks system call `taskSpawn`. When the task is activated, it calls the downstream function call subsystem code. The task is deleted using `taskDelete` during model termination.
- A semaphore is created to synchronize the connected subsystem to the execution of the Task Synchronization block.
- The spawned task is wrapped in an infinite `for` loop. In the loop, the spawned task listens for the semaphore, using `semTake`. When `semTake` is first called, `NO_WAIT` is specified. This allows the task to determine whether a second `semGive` has occurred prior to the completion of the function call

subsystem. This would indicate that the interrupt rate is too fast or the task priority is too low.

- The Task Synchronization block generates synchronization code (e.g., `semGive()`). This code allows the spawned task to run; the task in turn calls the connected function call subsystem code. The synchronization code can run at interrupt level. This is accomplished by connecting the Task Synchronization block to the output of an Async Interrupt block, which triggers execution of the Task Synchronization block within an ISR.
- If blocks in the downstream algorithmic code require absolute or elapsed time, it can be supplied either by the timer maintained by the Async Interrupt block, or by an independent timer maintained by the task associated with the Task Synchronization block.

Task Synchronization Block Parameters

The figure below shows the VxWorks Task Synchronization block dialog box.



The Task Synchronization block parameters are

- **Task name:** The task name is passed as the first argument to the VxWorks `taskSpawn` system call. This name is used as the task function name by VxWorks. It also provides a debugging aid; routines use the task name to identify the task from which they are called.

- **Task priority:** The VxWorks task priority to be assigned to the function call subsystem task when it is spawned. VxWorks priorities range from 0 . . 255, with 0 representing the highest priority.
- **Stack size (bytes):** Maximum size to which the task's stack can grow. The stack size is allocated when the task is spawned by VxWorks. The stack size should be chosen based on the number of local variables in the task. You should determine the size by examining the generated code for the task (and all functions that are called from the generated code).
- **Synchronize the data transfer of this task with the caller task:** When this option is deselected (the default),
 - The Task Synchronization block maintains a timer that provides absolute or elapsed time values required by the computations of downstream blocks. This timer is independent of the timer maintained by the Async Interrupt block that calls the Task Synchronization block.
 - The **Timer resolution** field is displayed. This field determines the resolution of the Task Synchronization block's timer. By default, the timer value is obtained via the VxWorks tickGet call. The default resolution is 1/60 second. The tickGet resolution for your BSP might be different. You should determine the tickGet resolution for your BSP and enter it in the **Timer resolution** field.
 - The timer value is read at the time that the task associated with the Task Synchronization block is activated by VxWorks. Data transfers to blocks called by the Task Synchronization block execute within this task.

When **Synchronize the data transfer of this task with the caller task** is selected

- The Task Synchronization block does not maintain an independent timer, and the **Timer resolution** field is not displayed.
- Downstream blocks that require timers use the timer maintained by the Async Interrupt block that calls the Task Synchronization block (see "Using Timers in Asynchronous Tasks" on page 16-29). The timer value is read at the time the asynchronous interrupt is serviced, and data transfers to blocks called by the Task Synchronization block execute within the task associated with the Async Interrupt block. Therefore, data transfers are said to be synchronized with the caller.

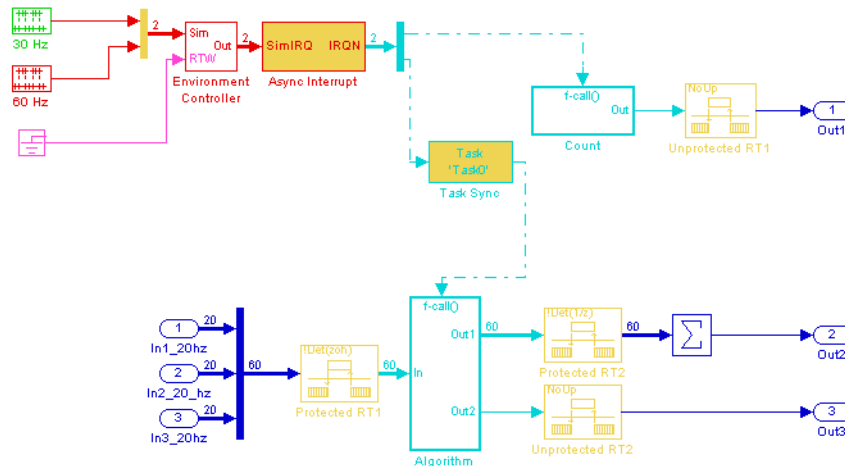
When you design your application, consider when timer and signal input values should be taken for the downstream function call subsystem that is

connected to the Task block. By default (**Synchronize the data transfer of this task with the caller task** is deselected), the time and input data are read when the task is activated by VxWorks. For this case, the data (inputs and time) are said to be synchronized to the task itself.

When **Synchronize the data transfer of this task with the caller task** is selected, and the Task block is driven by an Async Interrupt block, the time and input data are read when the interrupt occurs (i.e., within the ISR). For this case, data is said to be synchronized to the caller of the Task block.

Task Synchronization Block Example

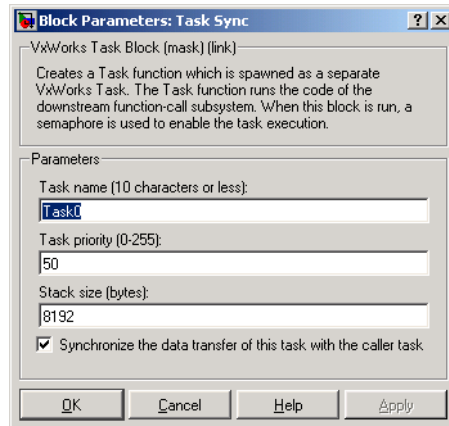
This section examines the use of the Task Synchronization block in the `rtdemo_async` model. The block diagram is shown below. Before reading the discussion, open the demo model and double-click the **Generate Code** button. You can then examine the generated code in the HTML code generation report produced by the demo.



The asyncdemo Model

In this model, the Async Interrupt block is configured for VME interrupts 1 and 2, using interrupt vector offsets 192 and 193. Interrupt 2 is connected to the

Task Synchronization block (TaskSync), which in turn drives the Algorithm subsystem. The figure below shows the block parameters for TaskSync.



Initialization Code. The Task Synchronization block generates an initialization function *model_subsystem_Start* (in this demo called *rtwdemo_async_Algorithm_Start()*). This function is called during initialization by *MdlStart*, which itself creates and initializes the synchronization semaphore. It also spawns an independent task (*task0*).

```
.
.
rtwdemo_async_Algorithm_Start();

/* VxWorks Task Block: <S5>/S-Function (vxtask1) */
/* Spawn task: Task0 with priority 50 */
if ((* (SEM_ID *)rtwdemo_async_DWork.SFunction_PWORK.SemID =
    semBCreate(SEM_Q_PRIORITY, SEM_EMPTY)) == NULL) {
    printf("semBCreate call failed for block Task0.\n");
}
if ((rtwdemo_async_DWork.SFunction_IWORK.TaskID = taskSpawn("Task0",
    50.0, VX_FP_TASK, 8192.0, (FUNCPTR)Task0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0)) == ERROR) {
    printf("taskSpawn call failed for block Task0.\n");
}
.
```

After spawning Task0, Md1Start connects and enables the ISR (isr_num2_vec193) for interrupt 2:

```
/* 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);
```

The ordering of these operations is significant. The task must be spawned before the interrupt that activates it can be enabled.

Task Code and Task Synchronization. The function Task0, generated by the Task Synchronization block, runs as a VxWorks task. The task waits for a synchronization semaphore in an infinite for loop. If it obtains the semaphore, it updates its task timer and calls the Algorithm subsystem.

Note that for this demo, the **Synchronize the data transfer of this task with the caller task** option of the Task Synchronization block is selected. Therefore, the timer associated with the Task Synchronization block (rtM->Timing.clockTick3) is updated with the value of the timer that is maintained by the Async Interrupt block (rtM->Timing.clockTick4). Therefore, blocks within the Algorithm subsystem use timer values based on the time of the most recent interrupt (not the most recent activation of Task0).

```
/* VxWorks Task Block: <S5>/S-Function (vxtask1) */
/* Spawned with priority: 50 */
void Task0(void)
{
    /* Wait for semaphore to be released by system:
rtwdemo_async/Task Sync */
    for(;;) {
        if (semTake(*(SEM_ID
*)rtwdemo_async_DWork.SFunction_PWORK.SemID,NO_WAIT) !=
ERROR) {
            logMsg("Rate for Task Task0() too fast.\n",0,0,0,0,0,0);
#ifdef STOPNOVERRUN
            logMsg("Aborting real-time simulation.\n",0,0,0,0,0,0);
            semGive(stopSem);
#endif
        }
    }
}
```

```

        return(ERROR);
    #endif
    } else {
        semTake(*(SEM_ID
*)rtwdemo_async_DWork.SFunction_PWORK.SemID,
        WAIT_FOREVER);
    }
    /* Use the upstream clock tick counter for this Task. */
    rtwdemo_async_M->Timing.clockTick3 =
rtwdemo_async_M->Timing.clockTick4;
    /* Call the system: <Root>/Algorithm */

    rtwdemo_async_Algorithm(0, 4);
}
}

```

The semaphore is granted by the function `isr_num2_vec193`, which is called from interrupt level:

```

/* 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 */
    rtwdemo_async_M->Timing.clockTick4 = tickGet();

    /* Call the system: <S4>/Subsystem */

    /* Output and update for function-call system: '<S4>/Subsystem' */
    {

        /* VxWorks Task Block: <S5>/S-Function (vxtask1) */
        /* Release semaphore for system task: Task0 */
        semGive(*(SEM_ID *)rtwdemo_async_DWork.SFunction_PWORK.SemID);
    }
}

```

The ISR maintains a timer that stores the tick count at the time of interrupt. This timer is updated before releasing the semaphore that activates Task0.

As this example shows, the Task Synchronization block generates only a small amount of interrupt-level code.

Task Termination. The Task Synchronization block also generates the following termination code.

```
/* Model terminate function */
void rtwdemo_async_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);

    /* Terminate for function-call system: '<S4>/Subsystem' */
    if(rtwdemo_async_M != NULL) {

        /* VxWorks Task Block: <S5>/S-Function (vxtask1) */
        /* Destroy task: Task0 */
        taskDelete(rtwdemo_async_DWork.SFunction_IWORK.TaskID);
    }
}
```

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/or from the function call subsystem can be in the process of being written or read 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 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 Chapter 8, “Models with Multiple Sample Rates.”

In a completely periodic model, you can handle rate transition issues transparently by selecting the **Automatically handle data transfers between tasks** option on the **Solver** page of the **Configuration Parameters** dialog box. This saves you from having to manually insert rate transition blocks in order to avoid illegal rate transitions in multirate periodic models. The inserted blocks are configured to ensure that data is transferred deterministically and that data integrity is maintained during the transfers.

However, automatic handling of rate transitions is currently supported only in transitions between two different *periodic* rates. Therefore, Rate Transition blocks must be manually inserted into your models when

- An asynchronously executed subsystem has inputs or outputs connected to a periodically executed block. This is called an *asynchronous-to-periodic* rate transition.
- An asynchronously executed subsystem has inputs or outputs connected to another asynchronously executed subsystem. This is called an *asynchronous-to-asynchronous* rate transition.

Determinism cannot be guaranteed in such cases. Therefore, when you insert Rate Transition blocks in the either situation, you need to deselect the **Ensure data determinism** check box in the block dialog.

When a Rate Transition block is inserted between two blocks to ensure data integrity, the priority of the tasks associated with both blocks must be known, so that the Rate Transition block can configure itself correctly for either a higher-to-lower priority transition or a lower-to-higher priority transition.

Priorities of periodic tasks are assigned by Simulink, in accordance with the options specified in the **Solver options** section of the **Solver** pane of the **Configuration Parameters** dialog. 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 via the **Periodic sample time properties** field. Simulink 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 **Effective 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.

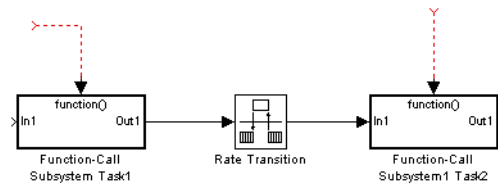
For the Task Synchronization block, the **Task priority** field similarly specifies the block priority relative to connected blocks (in addition to assigning a VxWorks priority to the generated task code). If VxWorks is the target, the **Higher priority value indicates higher task priority** option should be deselected.

The VxWorks 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.

Handling Rate Transitions Between Asynchronous and/or Periodic Tasks

Rate transitions between tasks can be legal or illegal depending on the nature of the two tasks involved. In many instances, data integrity or determinism cannot be ensured. The following model and table illustrate what options are possible in specific cases:



Possible Rate Transition Solutions Between Two Function Call Subsystems

	Task2 Async Priority = 1	Task2 Async Priority = 2	Task2 Async Priority Unspecified	Task2 Periodic Priority = 1	Task2 Periodic Priority = 2
Task1 Async Priority = 1	A2 * B1	A2 B1	A1 B1	A2 * B1	A2 B1
Task1 Async Priority = 2	A2 B1	A2 * B1	A1 B1	A2 B1	A2 * B1
Task1 Async Priority unspecified	A1 B1	A1 B1	A1 B1	A1 B1	A1 B1
Task1 Periodic Priority = 1	A2 * B1	A2 B1	A1 B1		A3
Task1 Periodic Priority = 2	A2 B1	A2 * B1	A1 B1	A3	

A1: One mode of built-in rate transition block (Unprotected) is allowed.

A2: Two modes of built-in rate transition block are allowed:

- Ensure data integrity, no determinism

- Unprotected

A3. Three modes of built-in rate transition block are allowed:

- Ensure data integrity and determinism
- Ensure data integrity, no determinism
- Unprotected

B1: Target-specific rate transition blocks can be used. The following are available for VxWorks:

- Read / Write pairs
- Unprotected

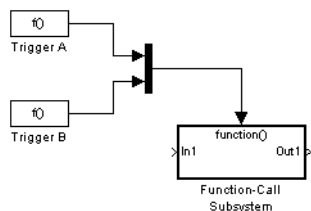
* If the **Asynchronous triggers with equal priority** option in the **Diagnostics** pane of the **Configuration Parameters** dialog is set to warning, you are advised as follows:

Asynchronous tasks associated with S-functions <task1> and <task2> are assigned the same priority: <priority>. Either modify these S-functions to specify differing priorities, or set the Asynchronous triggers with equal priority option in the Diagnostics page of the Configuration Parameters Dialog to none if tasks of equal priority cannot preempt each other in the target system.

Empty cells in the above table represent unsupported rate transition types.

Handling 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 **Asynchronous triggers with equal priority** option in the **Diagnostics** pane of the **Configuration Parameters** dialog; 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.

Real-Time Workshop 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 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 (e.g., 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 `BASE_RATE`, Real-Time Workshop generates timer code in the function call subsystem, updating the clock tick variable from the base rate clock tick. Data integrity is ensured if the same priority is assigned to TriggerA and TriggerB.
- If the time source is `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 is established according to the **Application lifespan** setting and the timer resolution set by the TriggerA and TriggerB S-functions (which must be the same). See “Using Timers in Asynchronous Tasks” on page 16-29 for more information.

Using Timers in Asynchronous Tasks

An ISR can set a source for absolute time. This is done with the function `ssSetTimeSource`, which has the following three options:

- **SELF**: Each generated ISR maintains its own absolute time counter, which is distinct from any 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 or elapsed time values required by block computations.
- **CALLER**: The ISR reads time from a counter maintained by its caller. Time resolution is thus the same as its caller's resolution.
- **BASE_RATE**: The ISR can read absolute time from its base rate. Time resolution is thus the same as its base rate resolution.

By default, the counter is implemented as a 32 bit unsigned integer member of the Timing substructure of the real-time model structure. For any target that supports the `rtModel` data structure, the counter word size is determined by the **Application lifespan (days)** 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;
};
```

Note that the example omits unused fields in the Timing data structure (a feature of ERT target code not found in GRT). For any target that supports the `rtModel` data structure, the counter word size is determined by the

Application lifespan (days) model parameter. Note, however, that if time source is set to `BASE_RATE`, the maximum length is two `UINT32` words, but if the time source is `SELF`, the maximum length is only one `UINT32`.

By default, the VxWorks library blocks set the timer source to `SELF` and update their counters via the system call `tickGet`. `tickGet` returns a timer value maintained by the VxWorks 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 targeting VxWorks, you can obtain better timer resolution by replacing the `tickGet` call and accessing a hardware timer via your BSP instead.

If you are implementing a custom asynchronous block for an RTOS other than VxWorks, you should either generate an equivalent call to the target RTOS, or generate code to read the appropriate 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. Note that 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 16-31.

Creating a Customized Asynchronous Library

This section describes how to implement asynchronous blocks for use with your target RTOS, using the Async Interrupt and Task Synchronization blocks as a starting point. (Rate Transition blocks are target independent, so you do not need to develop customized rate transition blocks.)

You can customize the asynchronous library blocks by modifying the block implementation. These files are

- The block's underlying S-function C MEX-file
- The TLC files that control code generation of the block

In addition, you need to modify the block masks to remove VxWorks references and better represent the parameters of your target RTOS.

Custom block implementation is an advanced topic, requiring familiarity with the Simulink C MEX S-function format and API, and with the Target Language Compiler (TLC). These topics are covered in the following documents:

- The Writing S-Functions document describes C MEX S-functions and the S-function API in general.
- The Target Language Compiler documentation and the “Writing S-Functions for Real-Time Workshop” chapter of the Real-Time Workshop documentation describe how to create a TLC block implementation for use in code generation.

The sections below discuss the 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`:

- `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 16-36.

C-MEX Block Implementation

Most of the code in `vxinterrupt1.c` performs ordinary functions that are not related to asynchronous support (e.g., obtaining and validating parameters from the block mask, marking parameters non tunable, 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.

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 effective priority for blocks executing at each interrupt level, as specified in the block’s **Effective 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`. Note that `SS_OPTION_ASYNCCHRONOUS_INTERRUPT` should be used when a function call subsystem is attached to an interrupt. For further information, see the documentation for `SS_OPTION` and `SS_OPTION_ASYNCCHRONOUS` in `matlabroot/simulink/include/simstruc.h`

```

ssSetOptions( S, (SS_OPTION_EXCEPTION_FREE_CODE |
                 SS_OPTION_DISALLOW_CONSTANT_SAMPLE_TIME |
                 SS_OPTION_ASYNCCHRONOUS_INTERRUPT |

```

TLC Implementation

This section discusses each function of `vxinterrupt1.tlc`, with an emphasis on target-specific features that you will need to change in order to generate code for your target RTOS.

Generating #include Directives. `vxinterrupt1.tlc` begins with the statement

```
%include "vxlib.tlc"
```

`vxlib.tlc` is a target-specific file that generates directives to include VxWorks header files. 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, there is no requirement to use this naming convention.

The function names are cached for use by the Outputs function, which generates the actual ISR code.

Outputs Function. Outputs iterates over all 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 C 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 Synchronization block. (This information is obtained via the `asynclib` calls `LibGetFcnCallBlock` and `LibGetBlockAttribute`.) If so,
 - The preemption flag for the ISR must be set to 1. An error results otherwise.
 - 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, `LibSetAsyncCounter` is called to generate a VxWorks `tickGet` function call and update the appropriate counter. In your implementation, you should generate either an equivalent call to the target RTOS, or generate code to read the appropriate timer register on the target hardware.

Note that if you are targeting VxWorks, you can obtain better timer resolution by replacing the `tickGet` call and accessing a hardware timer via your BSP instead. `tickGet` supports only a 1/60 second resolution.

Start Function. The Start function generates the required VxWorks calls (`int_connect` and `sysInt_Enable`) to connect and enable each ISR. You should replace this with appropriate calls to your target RTOS.

Terminate Function. The Terminate function generates the VxWorks call `sysIntDisable` to disable each ISR. You should replace this with appropriate calls to your target RTOS.

Task Synchronization Block Implementation

The source files for the Task Synchronization block are located in `matlabroot\rtw\c\tornado\devices`. They are

- `vxtask1.c`: C 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 16-36.

C-MEX Block Implementation

Like the Async Interrupt block, the Task Synchronization 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 **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

Generating #include Directives. `vxtask1.tlc` begins with the statement

```
%include "vxlib.tlc"
```

`vxlib.tlc` is a target-specific file that generates directives to include VxWorks header files. 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 identifier strings 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.

BlockInstanceData. The `BlockInstanceData` function generates storage for the semaphore that is used in management of the task spawned by the Task Synchronization block. Depending on the code format of the target, either a static storage declaration or a dynamic memory allocation call is generated. The ERT target and derived targets use a static memory declaration; the VxWorks target uses `malloc`.

Start Function. The Start function generates the required VxWorks calls to create a semaphore (`semBCreate`) and spawn a VxWorks task (`taskSpawn`). You should replace these with appropriate calls to your target RTOS.

Outputs Function. The Outputs function generates a VxWork 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 See "Task Synchronization Block Example" on page 16-18. Outputs also generates code (called from interrupt level) that grants the semaphore.

Terminate Function. The Terminate function generates the VxWorks call `taskDelete` to end execution of the task spawned by the block. You should replace this with appropriate calls to your target RTOS.

Note also that if the target RTOS has dynamically allocated any memory associated with the task (see "BlockInstanceData" on page 16-36), 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 (e.g., 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
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.
LibBlockExecuteFcnCall	For use by inlined S-functions with function call outputs. Generates code to execute a function call subsystem. LibBlockExecuteFcnCall calls the lower-level function LibExecuteFcnCall, but has a simplified argument list. See the TLC Reference Guide documentation for more information on LibExecuteFcnCall.
LibNeedAsyncCounter	If the calling block requires an asynchronous counter, returns TLC_TRUE, otherwise returns TLC_FALSE.
LibSetAsyncCounter	Generates code to set the tick value of the block's asynchronous counter.
LibGetCallerClockTickCounter	This function is used to access an upstream asynchronous task's time counter.

Data Exchange APIs

This chapter provides information on Real-Time Workshop application programming interfaces (APIs) that support data exchange interfaces between model code and other software components:

C-API for Interfacing with Signals and Parameters (p. 17-2)	Guidelines for use of the Real-Time Workshop signal monitoring and parameter tuning APIs
Creating an External Mode Communication Channel (p. 17-18)	How to support external mode on your custom target, using your own low-level communications layer
Combining Multiple Models (p. 17-29)	Strategies for combining several models (or several instances of the same model) into a single executable

C-API for Interfacing with Signals and Parameters

Simulink external mode offers a quick and easy way to monitor signals and modify parameter values while generated model code executes. (See Chapter 6, “External Mode.”) However, external mode might not be appropriate for your target or optimal for your application. In other cases, you might prefer to write your own code to access parameters and signals of a model directly, rather than using the external mode mechanism.

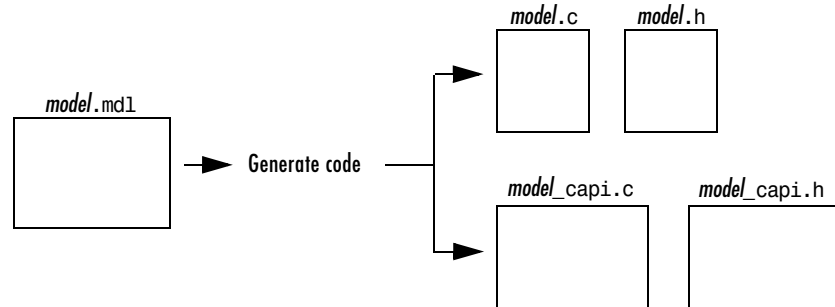
Real-Time Workshop supports several approaches to the task of interfacing block parameters and signals to your hand-written code. The **Model Parameter Configuration** dialog enables you to declare how the generated code allocates memory for variables used in your model. This allows your supervisory software to read or write block parameter variables as your model executes. See “Parameters: Storage, Interfacing, and Tuning” on page 5-2 for details on how to operate this dialog. Similarly, the **Signal Properties** dialog enables you to interface selected signals within your model. See “Signal Storage, Optimization, and Interfacing” on page 5-19 for details on how to operate this dialog.

In addition, the MathWorks provides the C-API. This is a target-based Real-Time Workshop feature that provides access to global block outputs and global parameters in the generated code. Using C-API, you can build target applications that log signals, monitor signals, and tune parameters while the generated code executes.

The C-API is designed to provide a smaller memory footprint. This is achieved by mapping information common to signals and parameters in smaller structures. An index into the structure map is provided in the actual signal or parameter structure. This allows the sharing of data across signals and parameters.

When you select the C-API feature and generate code, Real-Time Workshop generates two additional files, *model_capi.c* and *model_capi.h*, where *model* is the name of the model. Real-Time Workshop places the two C-API files in the build directory, based on settings on the **Configuration Parameters** dialog. The *model_capi.c* file contains information about global block signals and global parameters defined in the generated code. The *model_capi.h* 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. The generated files are illustrated below.



Generated Files with C-API Selected

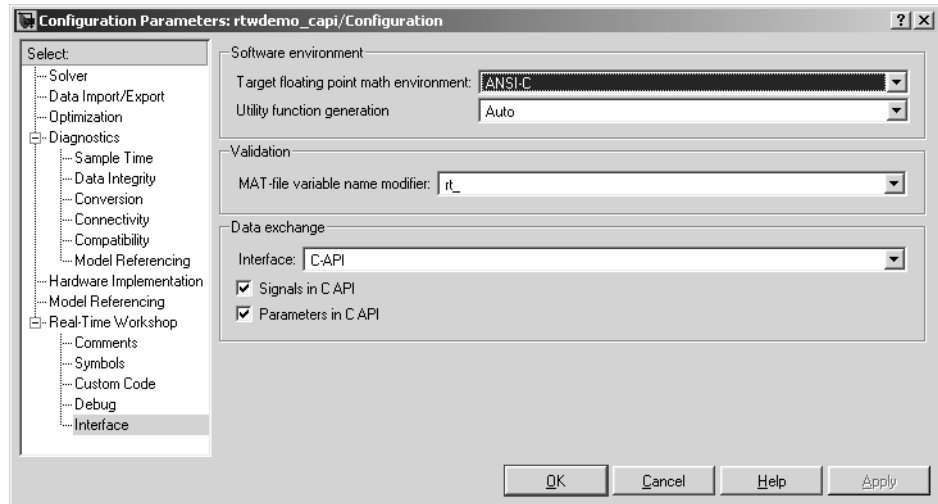
Generating the C-API Files

There are two ways to select the C-API feature: using the **Configuration Parameters** dialog box or directly from the MATLAB command line.

Selecting C-API with the Configuration Parameters Dialog Box

- 1 In the open model, select **Configuration Parameters** on the **Simulation** menu. The **Configuration Parameters** dialog box appears.
- 2 Click **Interface** under **Real-Time Workshop** on the left pane. The **Interface** pane appears on the right.
- 3 Select C-API in the **Interface** field. The **Signals in C API** and **Parameters in C API** check boxes appear, as shown below.
- 4 If you want to generate C-API for global block outputs, select the **Signals in C API** check box. If you want to generate C-API for global block and model parameters, select the **Parameters in C API** check box. If you select both check boxes, the default, both signals and parameters will appear in the C-API.
- 5 Click the **Apply** button.
- 6 Click **Real-Time Workshop** in the left pane. The **Generate code** button appears in the right pane.

7 Click Generate Code.



C-API Checkboxes on Configuration Parameters Dialog Box

Selecting C-API from the MATLAB Command Line

From the MATLAB command line you can select or clear the two C-API checkboxes on the **Configuration Parameters** dialog using the `uset_param` command. Type one or more of the following commands on the MATLAB command line as desired, where `modelName` is the one-word name of the model:

To select **Signals in C API**, type

```
uset_param(modelName, 'RTWCAPISignals', 'on')
```

To clear **Signals in C API**, type

```
uset_param(modelName, 'RTWCAPISignals', 'off')
```

To select **Parameters in C API**, type

```
uset_param(modelName, 'RTWCAPIParams', 'on')
```

To clear **Parameters in C API**, type

```
uset_param(modelName, 'RTWCAPIParams', 'off')
```

Description of C-API Files

The following sections describe the content of the generated `model_capi.c` file and other relevant code, where `model` is the name of your model. Of course, code you generate from your model might be different.

Note You can find a demo model, `rtwdemo_capi.mdl`, illustrating the use of the C-API in `matlabroot/toolbox/rtw/rtwdemos`. To open it, type `rtwdemo_capi` in MATLAB.

The C-API C File

The `model_capi.c` file contains data structures that describe the model's signals and parameters. `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's path, name, address, data type, dimensions, and fixed-point information.

`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's name, block path, address, data type, dimension, and fixed-point information. Note that if the **Inline parameters** check box on the **Optimization** pane of the **Configuration Parameters** dialog is selected, Real-Time Workshop generates an empty `rtBlockParameters`.

`rtModelParameters` is an array of structures that contains information about all workspace variables that were referenced as block parameters by one or more blocks or Stateflow charts in the model. Each element in the array is of data type `rtwCAPI_ModelParameters`. Note that if the **Inline parameters** check box on the **Optimization** pane of the **Configuration Parameters** dialog is cleared, Real-Time Workshop generates elements of this array that correspond to Stateflow data of machine scope.

The map vector `rtDataAddrMap` is an array of base addresses of signals and parameters that appear in `rtBlockSignals`, `rtBlockParameters`, and `rtModelParameters` structures in the `model_capi.c` file. Each element of the `rtDataAddrMap` array is a pointer to void (`void*`).

The `rtDataTypeMap` array 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.

The `rtDimensionMap` array contains information about the various data dimensions in the model. Each element of this array is of type `struct rtwCAPI_DimensionsMap`. 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.

The `rtFixPtMap` array contains fixed-point information about the signals and parameters. Each element of this array is of type `struct rtwCAPI_FixPtMap`. The members of this structure provide information about the data's scaling, bias, exponent, and whether or not the fixed-point data is signed. If the model does not have fixed-point data (signal or parameter), Real-Time Workshop assigns NULL or zero values to the elements of the `rtFixPtMap` array in the `model_capi.c` file.

The `rtSampleTimeMap` array contains sampling information about the model's global signals. (This array contains no 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.

C-API Signals. To interface with a signal, an application typically requires the signal's

- Description (name and block path)
- Address
- Data type information (native data type, complexity, fixed-point information)
- Dimensions
- Sampling and logging information

The C-API provides `rtBlockSignals`, which is an array of structures of type `struct rtwCAPI_Signals`. The `rtwCAPI_Signals` structure is defined in `matlabroot/rtw/c/src/model_capi.h`. The `rtwCAPI_Signals` structure captures the signal's description, address, data type information, dimensions, and sampling information.

Here is the section of code in *model_capi.c* that provides information on C-API signals:

```

15  /* Block output signal information */
16  static const rtwC_API_Signals rtBlockSignals[] = {
17
18      /* addrMapIndex, blockPath,
19       * signalLabel, portNumber, dataTypeIndex, dimIndex,
20       * fxpIndex, sampTimeIndex
21       */
22
23      {0, "rtwdemo_capi/Gain1",
24       "NULL", 0, 0, 0, 0, 0},
25
26      {1, "rtwdemo_capi/Gain3",
27       "sig1_sg", 0, 0, 0, 0, 0},
28
29      {2, "rtwdemo_capi/lu2d",
30       "sig2_eg", 0, 0, 1, 0, 0},
31
32      {
33      0, NULL, NULL, 0, 0, 0, 0, 0
34      }
35  };

```

Note To better understand the code, be sure to read the file's comments. Notice the comment that begins on line 18, above, for example. This comment lists the members of the `rtwC_API_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 `signalLabel` is the third member of the structure. Lines 29 and 30 describe the third signal. Thus, from line 30, you infer that this signal name (label) is `sig2_eg`.

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. Take the second signal, described by the code in lines 26 and 27, for example:

```
26     {1, "rtwdemo_capi/Gain3",  
27     "sig1_sg", 0, 0, 0, 0, 0}
```

This signal, named `sig1_sg`, is the output signal of the first port of the block `rtwdemo_capi/Gain3`. (It is the first port because the index for `portNumber` on line 27 is assigned the value 0.) The address of this signal is given by the `addrMapIndex`, which, in this example, is 1 on line 26. This gives the index to the `rtDataAddrMap` array, found later in `model_capi.c`.

The first member value on line 26 is 1. This is the value of `addrMapIndex` for the signal described on lines 26 to 27. The index of 1 points to the second element in the `rtDataAddrMap` array. So, from the `rtDataAddrMap` array, you can infer that the address of this signal is `&rtwdemo_capi_B.sig1_sg`.

This level of indirection is provided to support multiple code instances of the same model. Note that 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 “Interface Pane” and “Model Entry Points” in the Real-Time Workshop Embedded Coder documentation.)

The `dataTypeIndex` provides the index to the `rtDataTypeMap` array indicating the data type of the signal:

```
/* Data Type Map - Use TypeMapIndex to access this structure. */  
static const rtwCAPI_DataTypeMap rtDataTypeMap[]={  
    /* cName, mwName, numElements, elemMapIndex, dataSize,  
    isComplex */  
    {"double", "real_T", 0, 0, sizeof(real_T), SS_DOUBLE, 0}  
};
```

Because this index is 0, it points to the first structure element in the array. That is, it points to “{“double”, “real_T”, 0, 0, sizeof(real_T), SS_DOUBLE, 0}” above. So you can infer that the signal’s data type 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. This saves memory for each signal.

The `dimIndex` provides the index to the `rtDimensionMap` array indicating the dimensions of the signal:

Because this index is 0 on line 27, it points to the first element in the `rtDimensionMap` array:

```
/* dataOrientation, dimArrayIndex, numDims */
{rtwCAPI_SCALAR, 0, 2}
```

From this structure you can infer that this is a scalar signal having a dimension of 2.

The `fixPtIndex` provides the index to the `rtFixPtMap` array indicating any fixed-point information about the signal. Your code can use the scaling information provided 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” in the Simulink Fixed-Point documentation.)

Because this index is 0 on line 27, the signal has no fixed-point information. Note that a fixed-point map index of zero always means that the signal has no fixed-point information.

The `sampTimeIndex` provides the index to the `rtSampleTimeMap` array indicating task information about the signal. The sampling information can be useful if you log multirate signals or conditionally executed signals.

Note that `model_capi.c` includes `rtw_capi.h`. Any source file that references the `rtBlockSignals` array also should include `rtw_capi.h`.

C-API Parameters. Similar to signals, a typical parameter tuning or monitoring application requires a parameter’s

- Description
- Address
- Data type information
- Dimensions

The C-API provides `rtModelParameters`, which is an array of structures of type `struct rtwCAPI_ModelParameters`. The `rtwCAPI_ModelParameters` structure is defined in `matlabroot/rtw/c/src/model_capi.h`. The structure captures the parameter’s description, address, data type information, and dimensions.

Each element in the `rtModelParameters` array corresponds to a tunable parameter in the model. The code below is an example of elements in this array:

```
53 /* Tunable variable parameters */
54
55 static const rtwCAPI_ModelParameters rtModelParameters[] = {
56
57 /* addrMapIndex,varName,dataTypeIndex,dimIndex,fixPtIndex */
58 {3, "Ki", 0, 0, 0},
59 {4, "Kp", 0, 0, 0},
60 {5, "p1", 0, 2, 0},
61 {6, "p2", 0, 3, 0},
62 {7, "p3", 0, 4, 0},
63 {0, NULL, 0, 0, 0}
64};
```

Notice line 61, for example. The `varName` (variable name) of the line 61 parameter is `p2`. The address of this parameter is given by the `addrMapIndex` which, in this example, is 3. This is the index to the `rtDataAddrMap` array, found later in `model_capi.c`. Because the index is zero based, 6 corresponds to the seventh element in `rtDataAddrMap`, `&(p2)`. The `dataTypeIndex` is 0, which corresponds to a double and noncomplex parameter. The `dimIndex` (dimension index) is 3. So it points to the fourth element in the `rtDimensionMap` array. Later in the `model_capi.c` file this fourth element appears as

```
{rtwCAPI_Matrix_COL_MAJOR, 6, 2}
```

As mentioned in “The C-API C File” on page 17-5, `rtBlockParameters` is an array of structures of type `struct rtwCAPI_Parameters`. These structures are defined in `matlabroot/rtw/c/src/rtw_capi.h`. However, in the example, all the members of the structure `rtwCAPI_BlockParameters` are assigned `NULL` and zero values. This is because the **Inline parameters** check box on the **Optimization** pane of the **Configuration Parameters** dialog is selected. If you clear this check box, the block parameters are generated in the `rtwCAPI_BlockParameters` structure.

In this manner, the **Inline parameters** check box affects the information generated in the `rtBlockParameters` and `rtModelParameters` arrays.

If **Inline parameters** is cleared,

- The `rtBlockParameters` array contains an entry for every modifiable parameter of every block in the model.
- The `rtModelParameters` array contains only Stateflow data of machine scope. Real-Time Workshop assigns its elements only NULL or zero values in the absence of such data.

If **Inline parameters** is selected,

- The `rtBlockParameters` array is empty. Real-Time Workshop assigns its elements only NULL or zero values.
- The `rtModelParameters` array contains entries for all workspace variables that are referenced as tunable Simulink block parameters or Stateflow data of machine scope.

Mapping C-API Data Structures to Real-Time Model

The real-time model data structure encapsulates model data and associated information necessary to describe the model fully. For details, see “The Real-Time Model Data Structure” on page 7-30. When you select the C-API feature and generate code, Real-Time Workshop adds another member to the real-time model data structure:

```
struct {
    rtwCAPI_ModelMappingInfo mmi;
} DataMapInfo;
```

This member defines `mmi` (for model mapping information) of type `struct rtwCAPI_ModelMappingInfo`. The structure is provided 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 the `model_capi.c`.

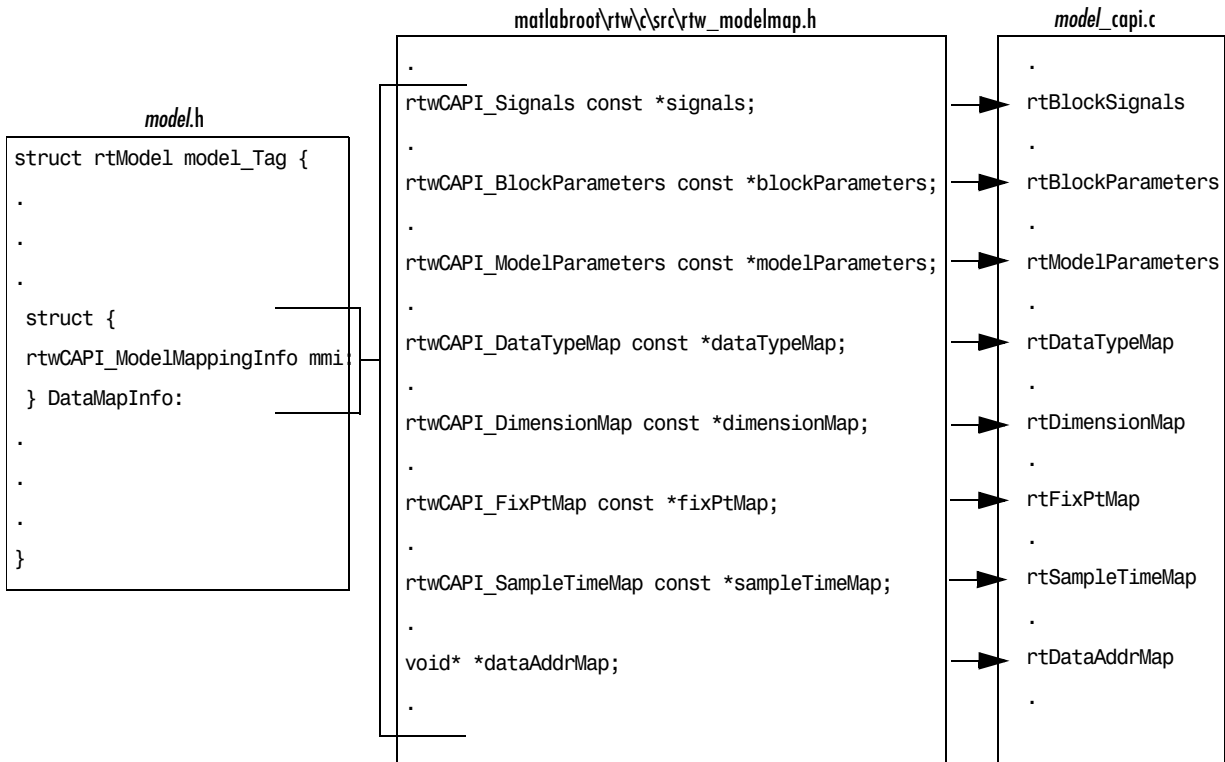
Initializing values of `mmi` members to the arrays accomplishes the mapping. See the figure below. Each member points to one of the arrays of structures in `model_capi.c`. For example, the address of the `rtBlockSignals` array of structures is allocated to the first member of the `mmi` substructure in `model.c`, using the following code in the `rtwmodelmap.h` file:

```
/* signals */
struct {
    rtwCAPI_Signals const *signals; /* Signals Array */
```

```
uint_T numSignals; /* Num Signals */
} Signals;
```

The model initialize function in *model.c* performs the initializing during model initialization. It does so by calling the C-API initialize function:

```
rtwdemo_capi_InitializeDataMapInfo(rtwdemo_capi_M).
```



Mapping between Model and C-API Arrays of Structures

Using the C-API in an Application

The C-API provides you with the flexibility of writing your own application code to interact with the signals and parameters. Your target-based application code is compiled with the Real-Time Workshop generated code into

an executable. The target-based application code accesses the C-API structure arrays in `model_capi.c` file. 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 `rtw_modelmap.h` file provides macros for accessing the structures in these arrays and their members.

An example application is provided below that prints the parameter values of all 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`.
- Uses `rtwCAPI_GetNumModelParameters` to get the number of model parameters in mapped C-API:

```
unit_T nModelParams = rtwCAPI_GetNumModelParameters(mmi);
```
- Uses `rtwCAPI_GetModelParameters` to access the array of all 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);
    }
}
```

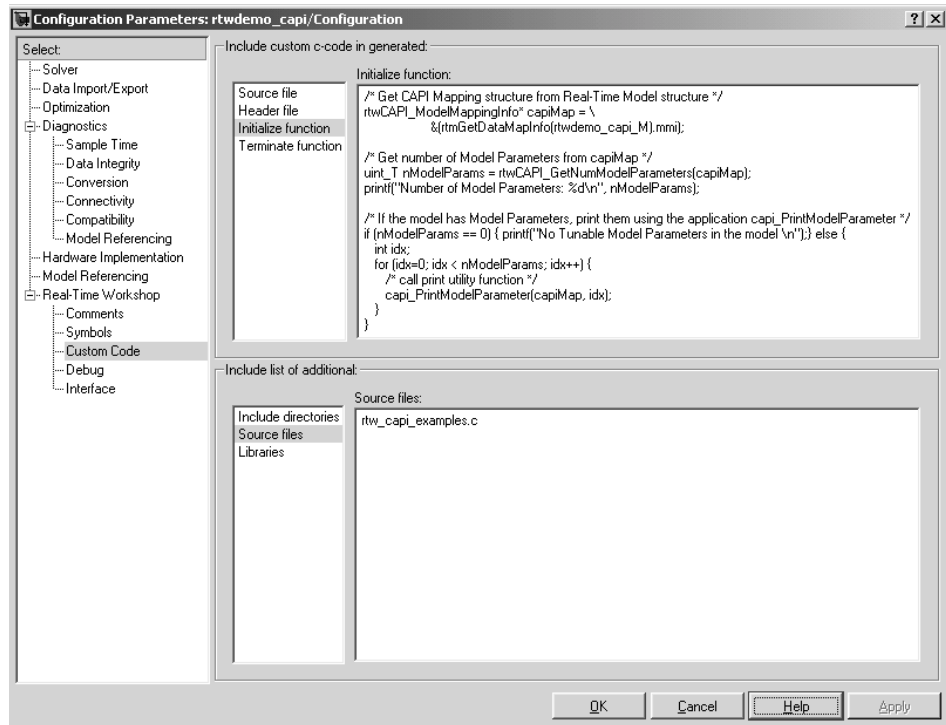
The print utility function is provided in `matlabroot/rtw/c/src/rtw_capi_examples.c`. This file contains utility functions for accessing the C-API structures.

To become familiar with the example code, try building a model that displays all the tunable block parameters and MATLAB variables the screen. You can use `rtwdemo_capi.mdl`, the C-API demo model, for this purpose:

- 1 In the open model `rtwdemo_capi.mdl`, select **Configuration Parameters** on the **Simulation** menu. The **Configuration Parameters** dialog box appears.
- 2 Select **Optimization** and then select the **Inline parameters** check box.
- 3 To embed your custom application code in the generated code, use the **Custom Code** pane on the **Configuration Parameters** dialog: Click **Custom Code** under **Real-Time Workshop** on the left pane, and then click **Initialize function** that appears in the center pane. The **Initialize function** pane appears on the right.
- 4 Type into this pane the example application code provided above. This embeds the application code in the `MdlStart` function. (Note that if you are using `ert.tlc`, the code appear in the function `model_initialize`.)
- 5 In the **Include list of additional** subpane, click **Source Files**, and type `rtw_capi_examples.c`. See the figure below.
- 6 Click the **Apply** button.
- 7 Click **Real-Time Workshop** in the left pane.
- 8 Clear the **Generate code only** check box. The **Build** button appears.

9 Click **Build**. Real-Time Workshop generates the executable.

10 Type `!rtwdemo_capi` at the MATLAB command line to run the executable. Parameter information is displayed in the Command Window.



C-API Limitations

The C-API feature has the following limitations.

- The following code formats are not supported:
 - S-function
 - Accelerator
- For ERT-based targets, the C-API requires that support for floating-point code be enabled.
- The following signals are not supported:
 - External inputs

- External outputs
- Local block outputs
- Parameters local to Stateflow are not supported.
- Regarding limitation of custom Sstorage classes:
 - Objects without the package `csc_registration` file are not supported.
 - `BitPackBoolean` objects, grouped custom storage classes, and objects defined via macro are not supported.
 - 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 only take effect in code generation if you use the ERT target and clear the **Ignore custom storage classes** check box on the **Configuration Parameters** dialog.

Target Language Compiler API for Signals and Parameters

Real-Time Workshop provides a 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 all 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 the Target Language Compiler documentation 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.

Creating an External Mode Communication Channel

This section helps you to connect your custom target via external mode using your own low-level communications layer. The following 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 Real-Time Workshop programs, and with the basic operation of external mode. These topics are described in Chapter 7, “Program Architecture,” and Chapter 6, “External Mode.”

The Design of External Mode

External mode communication between Simulink and a target system is based on a client/server architecture. The client (Simulink) 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 Simulink 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, GRT malloc, ERT, and RSim targets support host/target communication via TCP/IP and RS-232 (serial) communication. The Tornado target supports TCP/IP only. Note that serial transport is implemented only for Windows 32-bit architectures.

Real-Time Workshop provides full source code for both the client and server-side external mode modules, as used by the GRT, GRT malloc, ERT, Rapid Simulation, Real-Time Windows, xPC, and Tornado targets. 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	Source File
TCP/IP client	<i>matlabroot/rtw/ext_mode/tcpip/ext_tcpip_transport.c</i>
TCP/IP server	<i>matlabroot/rtw/c/src/ext_mode/tcpip/ext_svr_tcpip_transport.c</i>
Serial client	<i>matlabroot/rtw/ext_mode/serial/ext_serial_transport.c</i>
Serial server	<i>matlabroot/rtw/c/src/ext_mode/serial/ext_svr_serial_transport.c</i>

The modules *ext_tcpip_transport.c* and *ext_serial_transport.c* implement the client-side transport functions. The modules *ext_svr_tcpip_transport.c* and *ext_svr_serial_transport.c* contain the corresponding server-side functions. You can edit copies of these files (but do not modify the originals). You can support external mode via your own low-level communications layer by creating similar files using the following templates:

- *matlabroot/rtw/ext_mode/custom/ext_custom_transport.c*
- *matlabroot/rtw/c/src/ext_mode/custom/ext_svr_custom_transport.c*

Note Do not modify working source files. Use the templates provided in the */custom* directory 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. Code provided by Real-Time Workshop handles these functions.

On the client (Simulink) side, communications are handled by *ext_comm* (for TCP/IP) and *ext_serial_win32_comm* (for serial) C MEX-files. This component is implemented as a DLL on Windows, or as a shared library on UNIX.

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. These modules, called from the main program and the model execution engine, are independent of the generated model code.

Creating a Custom Client Transport Protocol

To implement your own *client-side* low-level transport protocol,

- 1 Edit the template `ext_custom_transport.c` to replace low-level TCP/IP calls with your own communication calls.
- 2 Save it as `ext_xxx_transport.c` (replacing `xxx` with some identifier meaningful to you).
- 3 Generate a customized version of `ext_comm` using the `mex` command, replacing the entry `ext_tcpip_transport.c` with your custom component's filename.

See the table “Commands to Rebuild `ext_comm` MEX-Files” on page 17-27 for examples of this `mex` command.

- 4 Finally, to make your new transport layer available to users, modify `matlabroot/toolbox/simulink/simulink/extmode_transports.m` according to instructions within it to modify the **Configuration Parameters Interface** pane. The name of the transport will appear in the **Transport layer** menu, and the name of your customized transport file will occupy the noneditable **MEX-file name** field when you update the model and select your custom transport.

Creating a Custom Server Transport Protocol

To implement your own *server-side* low-level transport protocol,

- 1 Replace low-level TCP/IP calls in `ext_svr_xxx_transport.c` with your own communication calls.
- 2 Modify all the template makefiles that you use to support the new transport. 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 `ext_svr_xxx_transport.c` and other server code into your executable.

- 3 Define symbols and functions common to both the client and server sides in the file you create from the template `ext_custom_utils.c` (replacing `ext_tcpip_utils.c`).

External Mode Communications Overview

This section gives a high-level overview of how a Real-Time Workshop generated program communicates with Simulink in external mode. This description is based on the TCP/IP version of external mode that ships with Real-Time Workshop. At this level of detail, however, there are no differences between the TCP/IP and serial implementations.

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 Simulink must be executing. Note that this does not mean that the model code in the server system must be executing. The server can be waiting for Simulink to issue a command to start model execution.

The client and server communicate via 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, Simulink 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 Simulink via 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

Host MEX-file Interface Source Files

The source files for the MEX-file interface component are located in the directory `matlabroot/rtw/ext_mode`:

- `common/ext_comm.c`

This file is the core of external mode communication. It acts as a relay station between the target and Simulink. `ext_comm.c` communicates to Simulink via

a shared data structure, `ExternalSim`. It communicates to the target via 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.

- `tcpip/ext_tcpip_transport.c` (or `serial/ext_serial_transport.c`)
This file implements required transport layer functions. Note that `ext_tcpip_transport.c` includes `ext_tcpip_utils.c`, which is generated into the target directory and contains functions common to client and server sides; similarly, `ext_serial_transport.c` includes `ext_serial_utils.c`, which is also generated into the target directory. The version of `ext_tcpip_transport.c` shipped with Real-Time Workshop uses TCP/IP functions including `recv()`, `send()`, and `socket()`.
- `common/ext_main.c`
This file is a MEX-file wrapper for external mode. `ext_main.c` interfaces to Simulink via the standard `mexFunction` call. (See “External Interfaces/API” in the MATLAB online documentation for information on `mexFunction`.) `ext_main.c` contains a function dispatcher, `esGetAction`, that sends requests from Simulink 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 Simulink (via function pointers).

Note You do not need to customize `ext_convert` in order to implement a custom transport layer. However, it might be necessary to customize `ext_convert` for the intended target. For example, if the target represents the float data type in Texas Instruments (TI) format, `ext_convert` must be modified to perform a TI to IEEE conversion.

- `common/extsim.h`
This file defines the `ExternalSim` data structure and access macros. This structure is used for communication between Simulink 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.

Target (Server) Source Files

These files are part of the run-time interface and are linked into the `model.exe` executable. They are located within `matlabroot/rtw/c/src/ext_mode/`.

- `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.
- `tcpip/ext_svr_tcpip_transport.c`
This file implements required transport layer functions. Note that `ext_svr_tcpip_transport.c` includes `ext_tcpip_utils.c`, which contains functions common to client and server sides, and `ext_serial_transport.c` includes helper functions from `ext_serial_utils.c`. The version of `ext_svr_tcpip_transport.c` shipped with Real-Time Workshop uses TCP/IP functions including `recv()`, `send()`, and `socket()`.
- `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 (e.g., `grt_main.c`). It contains code that is called via the background tasks of `ext_svr.c` as well as code that is called as part of the higher priority model execution.
- `dt_info.h` and `model.dt`
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.

Other Files

- `common/ext_share.h`
Contains message code definitions and other definitions required by both the host and target modules.
- `tcpip/ext_tcpip_utils.c` and `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 TCP/IP and serial protocols, respectively.
- `ext_svr_transport.h`
This file defines functions that must be implemented by the transport layer.
- The serial transport implementation includes the additional files
 - `ext_serial_pkt.c` and `ext_serial_pkt.h`
 - `ext_serial_port.h`
 - `ext_serial_win32_port.c`

Guidelines for Implementing the Transport Layer

If you wish to implement a customized serial transport layer, the easiest way is to modify files provided with Real-Time Workshop. Copy and rename the file `ext_serial_win32_port.c` to `ext_serial_<your_transport>_port.c`, and modify it as needed. Then edit your new transport layer into `matlabroot/toolbox/simulink/simulink/extmode_transports.m` and change your template makefiles to point to your source files.

Requirements

- 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 appropriately.
- The target is assumed to support both `int32_T` and `uint32_T` data types.

Modifying `ext_tcpip_transport.c` and `ext_serial_transport.c`

Function prototypes in `common/ext_transport.h` define the calling interface for the host (client-side) transport layer functions. The implementations are in `ext_tcpip_transport.c` and `ext_serial_transport.c`. These, like other protocol-specific files described below, occupy parallel directories, `/tcpip` and `/serial`.

Host Transport Layer. To implement the host side of your transport layer,

- 1 Replace the functions in the “Visible Functions” section of template file `matlabroot/rtw/c/src/ext_mode/custom/ext_custom_transport.c` with functions that call your low-level communications primitives. The visible functions are called from other external mode modules such as `ext_comm.c`.

You must implement *all* the functions defined in template file `ext_custom_utils.c`, and those implementations must conform to the prototypes defined in `ext_custom_utils.c`.

- 2 Supply a definition for the `UserData` structure in `ext_tcpip_transport.c` and `ext_serial_transport.c`. This structure is required. If `UserData` is not necessary for your external mode implementation, define a `UserData` structure with one dummy field.
- 3 Replace the functions in `matlabroot/rtw/c/src/ext_mode/tcpip/ext_tcpip_utils.c` and/or `matlabroot/rtw/c/src/ext_mode/serial/ext_serial_utils.c` with functions that call your low-level communications primitives, or remove these functions. Functions defined in these files are common to the host and target, and are not part of the public interface to the transport layer.
- 4 Build the customized MEX-file executable using the MATLAB `mex` command. Do not replace the existing `ext_comm` MEX-file if you want to preserve its existing function. Instead, use the `-output` command option to name the resulting executable (e.g., `mex -output ext_myserial_comm ...` builds `ext_myserial_comm.dll` on Windows). Then edit `matlabroot/toolbox/simulink/simulink/extmode_transports.m` and change your template makefiles to make your new transport available.

The following table lists the commands for building the standard `ext_comm` module on PC and UNIX platforms.

Commands to Rebuild ext_comm MEX-Files

Platform	Commands
UNIX, TCP/IP	<pre>mex matlabroot/rtw/ext_mode/common/ext_comm.c matlabroot/rtw/ext_mode/common/ext_convert.c matlabroot/rtw/ext_mode/tcpip/ext_tcpip_transport.c -Imatlabroot/rtw/c/src/ext_mode/common -Imatlabroot/rtw/c/src/ext_mode/tcpip -Imatlabroot/rtw/ext_mode/common -output rtw/ext_comm</pre>
PC, TCP/IP	<pre>mex matlabroot\rtw\ext_mode\common\ext_comm.cmatlabroot\rtw\ext_mode\common\ext_convert.c matlabroot\rtw\ext_mode\tcpip\ext_tcpip_transport.c - Imatlabroot\rtw\c\src\ext_mode\common -Imatlabroot\rtw\c\src\ext_mode\tcpip -Imatlabroot\rtw\ext_mode\common -output rtw\ext_comm -DWIN32 <compiler_library_path>\wsock32.lib</pre>
PC, serial	<pre>mex matlabroot\rtw\ext_mode\common\ext_comm.c matlabroot\rtw\ext_mode\common\ext_convert.c matlabroot\rtw\ext_mode\serial\ext_serial_transport.c matlabroot\rtw\c\src\ext_mode\serial\ext_serial_pkt.c matlabroot\rtw\c\src\ext_mode\serial\ext_serial_win32_port.c -I<matlabroot>\rtw\c\src\ext_mode\common -Imatlabroot\rtw\c\src\ext_mode\serial -Imatlabroot\rtw\ext_mode\common -output rtw\ext_serial_win32 -DWIN32 <compiler_library_path>\wsock32.lib</pre>

See comments in the `ext_custom_transport.c` and `ext_custom_utils.c` source code modules for further details.

Note `mex` requires a compiler supported by the MATLAB API. See “External Interfaces/API” in the MATLAB online documentation for more information on the `mex` command.

Guidelines for Modifying `ext_svr_custom_transport`

The function prototypes in `ext_svr_transport.h` define the calling interface for the target (server) side transport layer functions. The implementations are in `ext_svr_tcpip_transport.c` and `ext_svr_serial_transport.c`.

To implement the target side of your transport layer,

- Supply code for the dummy functions in `ext_svr_custom_transport.c` to call your low-level communications primitives. These are the functions called from other target modules such as the main program. Use `ext_svr_tcpip_transport.c` and `ext_svr_serial_transport.c` for guidance as needed. You must implement all the functions defined in `ext_svr_transport.h`. Your implementations must conform to the prototypes defined in that file.
- Supply a definition for the `ExtUserData` structure in `ext_svr_transport.c`. This structure is required. If `ExtUserData` is not necessary for your external mode implementation, define an `ExtUserData` structure with one dummy field.
- Define the `EXT_BLOCKING` conditional in your implementation of `ext_svr_tcpip_transport.c` or `ext_svr_serial_transport.c` as needed:
 - Define `EXT_BLOCKING` as 0 to poll for a connection to the host (appropriate for single-threaded applications).
 - Define `EXT_BLOCKING` as 1 in multithreaded applications where tasks are able to block for a connection to the host without blocking the entire program.

See also the comments on `EXT_BLOCKING` in `ext_svr_transport.c`.

The `ext_svr_transport` source code modules are fully commented. See these files for further details.

Combining Multiple Models

If you want to combine several models (or several instances of the same model) into a single executable, Real-Time Workshop offers several options.

The most powerful — and usually simplest — solution is to create a “top” (or main) model, and to use Model blocks to include other models in it. Referenced models can themselves contain model blocks referencing other models. See “Generating Code from Models Containing Model Blocks” on page 4-18 for details on using Model blocks.

If the models to be combined are completely independent of one another and not hierarchically related, other approaches might be more appropriate.

When developing embedded systems using Real-Time Workshop Embedded Coder, you can interface the code for several models to a common harness program by directly calling the entry points to each model. However, Real-Time Workshop Embedded Coder target has certain restrictions that might not be appropriate for your application. For more information, see the Real-Time Workshop Embedded Coder documentation.

The GRT malloc target is another possible solution. Using it is appropriate in situations where you want do any or all of the following:

- Selectively control calls to more than one model
- Use dynamic memory allocation
- Include models that employ continuous states
- Log data to multiple files
- Run one of the models in external mode

To summarize by targets, your options are as follows:

Support for Combining Independent Multiple Models

GRT	No (except via Model blocks)
GRT Malloc	Yes
ERT	Yes
S-Function	No

Using GRT Malloc to Combine Models

This section discusses how to use the GRT malloc target to combine models into a single program. Before reading this section, you should become familiar with model execution in Real-Time Workshop programs. (See Chapter 7, “Program Architecture,” and Chapter 8, “Models with Multiple Sample Rates.”) It will be helpful to refer to `grt_malloc_main.c` while reading these chapters.

Building a multiple-model executable is fairly straightforward:

- 1 Generate and compile code from each of the models that are to be combined.
- 2 Combine the makefiles for each of the models into one makefile for creating the final multimodel executable.
- 3 Create a combined simulation engine by modifying `grt_malloc_main.c` to initialize and call the models correctly.
- 4 Run the combination makefile to link the object files from the models and the main program into an executable.

Sharing Data Across Models

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

Timing Issues

You must generate all the models you are combining with the same solver mode (either all singletasking or all multitasking.) In addition, if the models employ continuous states, the same solver should be used for all models.

Because each model has its own model-specific definition of the `rtModel` data structure, you must use an alternative mechanism to control model execution, as follows:

- The file `rtw/c/src/rtmcmacros.h` provides an `rtModel` API clue that can be used to call the `rt_OneStep` procedure.
- The `rtmcmacros.h` header file defines the `rtModelCommon` data structure, which has the minimum common elements in the `rtModel` structure required to step a model forward one time step.

- The `rtmcsetCommon` macro populates an object of type `rtModelCommon` by copying the respective similar elements in the model's `rtModel` object. Your main routine must create one `rtModelCommon` structure for each model being called by the main routine.
- The main routine will subsequently invoke `rt_OneStep` with a pointer to the `rtModelCommon` structure instead of a pointer to the `rtModel` structure.

If the base rates for the models are not the same, the main program (such as `grt_malloc_main`) must set up the timer interrupt to occur at the greatest common divisor rate of the models. The main program is responsible for calling each of the models at the appropriate time interval.

Data Logging and External Mode Support

A multiple-model program can log data to separate MAT-files for each model (as in the example program discussed below).

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

Blocks That Depend on Absolute Time

Some Simulink blocks use the value of absolute time (i.e., the time from the beginning of the program to the present time) to calculate their outputs. If you are designing a program that is intended to run indefinitely, then you must take care when using blocks that have a dependency on absolute time.

The problem arises when the value of time reaches the largest value that can be represented by the datatype used by the timer to store time. At that point, the timer overflows and the output of the block is no longer correct. If the target uses `rtModel`, you can avoid this by setting an appropriate **Application life span** option, see “Integer Timers in Generated Code” on page 15-3 for more information.

Note In addition to the blocks listed below, logging **Time** (in the **Data import/export** pane of the **Configuration parameters** dialog box) also requires absolute time.

The following Simulink blocks depend on absolute time:

- Backlash
- Chirp Signal
- Clock
- Derivative
- Digital Clock
- Discrete-Time Integrator (when used in triggered subsystems)
- From File
- From Workspace
- Pulse Generator
- Ramp

- Rate Limiter
- Repeating Sequence
- Scope
- Signal Generator
- SineWave
- Step
- To File
- To Workspace (only if logging to StructureWithTime format)
- Transport Delay
- Variable Transport Delay

Note The Sine Wave block is dependent on absolute time only when the **Sine type** parameter is set to Time-based. Set this parameter to Sample-based to avoid absolute time computations.

In addition to the Simulink block above:

- Blocks in other blocksets may reference absolute time. Please refer to the documentation for the blocksets that you use.
- Stateflow charts that use time are dependent on absolute time.

Generating ASAP2 Files

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

Overview (p. B-2)	Topics you should be familiar with before working with ASAP2 file generation
Targets Supporting ASAP2 (p. B-3)	Real-Time Workshop targets with built-in ASAP2 support
Defining ASAP2 Information (p. B-4)	Signal and parameter information from a Simulink model needed to create an ASAP2 file
Generating an ASAP2 File (p. B-6)	Procedure for creating an ASAP2 file from a Simulink model
Customizing an ASAP2 File (p. B-12)	Target Language Compiler (TLC) files you can change to customize the ASAP2 file generated from a Simulink model
Structure of the ASAP2 File (p. B-18)	Summary of the parts of the ASAP2 file and the Target Language Compiler functions used to write each part

Overview

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

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

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

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

Alternatively, you can access the demo by typing the following command at the MATLAB command prompt:

```
asap2demo
```

Targets Supporting ASAP2

ASAP2 file generation is available to all Real-Time Workshop target configurations. You can select these target configurations from the **System target file** browser. For example:

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

Procedures for generating ASAP2 files via these target configurations are given in “Generating an ASAP2 File” on page B-6.

Defining ASAP2 Information

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

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

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

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

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

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

Data Attribute	Defined In	Property Name
Data type	Model	Not applicable
Scaling (if fixed point data type)	Model	Not applicable

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
Minimum allowable value	Data object	Min
Maximum allowable value	Data object	Max
Units	Data object	DocUnits
Memory address (optional)	Data object (see note below)	MemoryAddress_ASAP2 (optional; see “Memory Address Attribute” below)

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

Generating an ASAP2 File

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

- Use the Generic Real-Time target or a Real-Time Workshop Embedded Coder target to generate an ASAP2 file as part of the code generation and build process.
- Use the ASAP-ASAP Data Definition target to generate only an ASAP2 file, without building an executable.

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

Generating ASAP2 Files Using the Generic Real-Time Target

The procedure for generating a model's data definition in ASAP2 format using the generic Real-Time target is as follows:

- 1 Create the desired model. Use appropriate parameter names and signal labels to refer to CHARACTERISTICS and MEASUREMENTS respectively.
- 2 Define the desired parameters and signals in the model to be `ASAP2.Parameter` and `ASAP2.Signal` objects in the MATLAB workspace.
- 3 For each ASAP2 data object, configure the `RTWInfo StorageClass` property to be one of the following:
 - `ExportedGlobal`
 - `ImportedExtern`
 - `ImportedExternPointer`

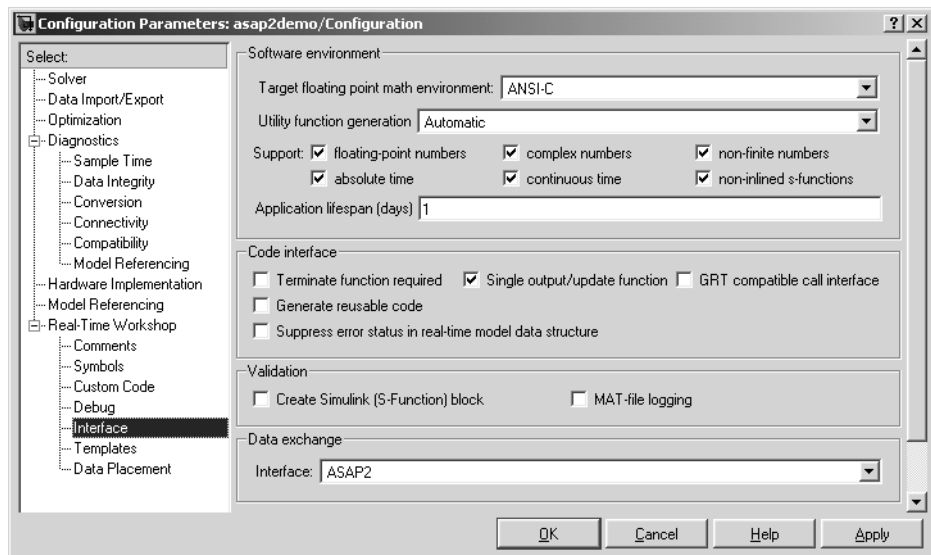
This configures the data objects so that their corresponding declarations in the generated code are unstructured global storage declarations.

- 4 Configure the remaining properties as desired for each ASAP2 data object.
- 5 On the **Optimization** pane of the **Configuration Parameters** dialog box, select the **Inline parameters** check box.

Note that you should *not* configure the parameters associated with your data objects as global (tunable) parameters in the **Model Parameter**

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

- 6 On the **Real-Time Workshop** pane, click **Browse** to open the **System target file** browser. In the browser, select Generic Real-Time Target and click **OK**.
- 7 In the **Interface** field on the **Interface** pane, select ASAP2.



- 8 Select the **Generate code only** checkbox on the **Real-Time Workshop** pane.
- 9 Click **Apply**.
- 10 Click **Generate code**.

Real-Time Workshop writes the ASAP2 file to the build directory. The ASAP2 filename is controlled by the ASAP2 setup file. By default, the file is named `modelName.a21`, where `modelName` is the name of the model.

Generating ASAP2 Files Using the Real-Time Workshop Embedded Coder

The procedure for generating a model's data definition in ASAP2 format using the Real-Time Workshop Embedded Coder is as follows:

- 1** Create the desired model. Use appropriate parameter names and signal labels to refer to CHARACTERISTICS and MEASUREMENTS respectively.
- 2** Define the desired parameters and signals in the model to be ASAP2.Parameter and ASAP2.Signal objects in the MATLAB workspace.
- 3** For each ASAP2 data object, configure the RTWInfo StorageClass property to be one of the following:
 - ExportedGlobal
 - ImportedExtern
 - ImportedExternPointer

This configures the data objects so that their corresponding declarations in the generated code are unstructured global storage declarations.

- 4** Configure the remaining properties as desired for each ASAP2 data object.
- 5** On the **Optimization** pane of the **Configuration Parameters** dialog box, select the **Inline parameters** checkbox.

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

- 6** On the **Real-Time Workshop** pane, click **Browse** to open the **System target file** browser. In the browser, select any Embedded Real-Time Target and click **OK**.
- 7** In the **Interface** field on the **Interface** pane, select ASAP2.
- 8** Select the **Generate code only** checkbox on the **Real-Time Workshop** pane.

9 Click **Apply**.

10 Click **Generate code**.

Real-Time Workshop writes the ASAP2 file to the build directory. The ASAP2 filename is controlled by the ASAP2 setup file. By default, the file is named `model.a21`.

Generating ASAP2 Files Using the ASAM-ASAP2 Data Definition Target

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

- 1 Create the desired model. Use appropriate parameter names and signal labels to refer to CHARACTERISTICS and MEASUREMENTS respectively.
- 2 Define the relevant ASAP2.Parameter and ASAP2.Signal objects in the MATLAB workspace.
- 3 For each ASAP2 data object, configure the RTWInfo StorageClass property to be one of the following:
 - ExportedGlobal
 - ImportedExtern
 - ImportedExternPointer

This configures the data objects so that their corresponding declarations in the generated code are unstructured global storage declarations.

- 4 Configure the remaining properties as desired for each ASAP2 data object.
- 5 On the **Optimization** pane of the **Configuration Parameters** dialog box, select the **Inline parameters** checkbox.

Note that you should *not* configure the parameters associated with your data objects as global (tunable) parameters in the **Model Parameter Configuration** dialog box. If a parameter that resolves to a Simulink data object is configured using the **Model Parameter Configuration** dialog box, the dialog box configuration is ignored.

- 6 On the **Real-Time Workshop** pane, click **Browse** to open the **System target file** browser. In the browser, select ASAM-ASAP2 Data Definition Target and click **OK**.

- 7** Select the **Generate code only** checkbox on the **Real-Time Workshop** pane.
- 8** Click **Apply**.
- 9** Click **Generate code**.

Real-Time Workshop writes the ASAP2 file to the build directory. The ASAP2 filename is controlled by the ASAP2 setup file. By default, the file is named `model.a21`.

Customizing an ASAP2 File

The Real-Time Workshop Embedded Coder provides a number of TLC files to enable you to customize the ASAP2 file generated from a Simulink model.

ASAP2 File Structure on the MATLAB Path

The ASAP2 related files are organized within the directories identified below:

- TLC files for generating ASAP2 files
The *matlabroot/rtw/c/tlc/mw* directory contains TLC files that generate ASAP2 files. These files are included by the selected **RTW system target file**. (See “Targets Supporting ASAP2” on page B-3.)
- ASAP2 target files
The *matlabroot/toolbox/rtw/targets/asap2/asap2* directory contains the ASAP2 system target file and other control files.
- ASAP2 class creation files
The *matlabroot/toolbox/rtw/targets/asap2/asap2/@ASAP2* directory contains the M-files that define the `ASAP2.Parameter` and `ASAP2.Signal` classes.
- Customizable TLC files
The *matlabroot/toolbox/rtw/targets/asap2/asap2/user* directory contains files that you can modify to customize the content of your ASAP2 files.
- ASAP2 templates
The *matlabroot/toolbox/rtw/targets/asap2/asap2/user/templates* directory contains templates that define each type of CHARACTERISTIC in the ASAP2 file.

Customizing the Contents of the ASAP2 File

The ASAP2 related TLC files enable you to customize the appearance of the ASAP2 file generated from a Simulink model. Most customization is done by modifying or adding to the files contained in the

`matlabroot/toolbox/rtw/targets/asap2/asap2/user` directory. This section refers to this directory as the `asap2/user` directory.

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

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

The procedure for customizing the ASAP2 file is as follows:

- 1** Make a copy of the `asap2/user` directory before making any modifications.
- 2** Remove the old `asap2/user` directory from the MATLAB path, or add the new `asap2/user` directory to the MATLAB path above the old directory. This ensures that MATLAB uses the new ASAP2 setup file, `asap2setup.tlc`.

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

- 3** Modify the static parts of the ASAP2 file. These include
 - Project and header symbols, which are specified in `asap2setup.tlc`
 - Static sections of the file, such as file header and tail, A2ML, MOD_COMMON, etc. These are specified in `asap2userlib.tlc`.
 - Specify the appearance of the dynamic contents of the ASAP2 file by modifying the existing ASAP2 templates, or by defining new ASAP2 templates. Sections of the ASAP2 file affected include
 - RECORD_LAYOUTS: modify appropriate parts of the ASAP2 template files.
 - CHARACTERISTICS: modify appropriate parts of the ASAP2 template files.For more information on modifying the appearance of CHARACTERISTICS, see “ASAP2 Templates” on page B-14.
 - MEASUREMENTS: These are specified in `asap2userlib.tlc`.
 - COMPU_METHODS: These are specified in `asap2userlib.tlc`.

ASAP2 Templates

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

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

- 1 Define a parameter group. See “Defining Parameter Groups” on page B-14.
- 2 Create a template definition file. See “Creating Template Definition Files” on page B-15.
- 3 Include the template definition file in the TLC path. The path is specified in the ASAP2 setup file, asap2setup.tlc.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Function to Write the CHARACTERISTIC. This function writes out the CHARACTERISTIC associated with this template. The function is called by the built-in functions involved in the ASAP2 file generation process. The function

name must be defined as shown, with the appropriate template name after the underscore.

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

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

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

Structure of the ASAP2 File

The following table outlines the basic structure of the ASAP2 file and describes the Target Language Compiler 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 <code>asap2main.tlc</code>	TLC File Containing Function Definition
File header	<code>%<ASAP2UserFcnWriteFileHead()></code>	<code>asap2userlib.tlc</code>
<code>/begin PROJECT ""</code>	<code>/begin PROJECT "%<ASAP2ProjectName>"</code>	<code>asap2setup.tlc</code>
<code>/begin HEADER ""</code> HEADER contents	<code>/begin HEADER "%<ASAP2HeaderName>"</code> <code>%<ASAP2UserFcnWriteHeader()></code>	<code>asap2setup.tlc</code> <code>asap2userlib.tlc</code>
<code>/end HEADER</code>	<code>/end HEADER</code>	
<code>/begin MODULE ""</code> MODULE contents:	<code>/begin MODULE "%<ASAP2ModuleName>"}</code>	<code>asap2setup.tlc</code> <code>asap2userlib.tlc</code>
- A2ML - MOD_PAR - MOD_COMMON ...	<code>%<ASAP2UserFcnWriteHardwareInterface()></code>	
Model-dependent MODULE contents:	<code>%<SLibASAP2WriteDynamicContents()></code> Calls user-defined functions:	<code>asap2lib.tlc</code>
- RECORD_LAYOUTS - CHARACTERISTICS - ParameterGroups - ModelParameters	<code>...WriteRecordLayout_TemplateName()</code> <code>...WriteCharacteristic_TemplateName()</code> <code>...WriteCharacteristic_Scalar()</code>	<code>user/templates/.</code> <code>..</code>
- MEASUREMENTS - ExternalInputs - BlockOutputs	<code>...WriteMeasurement()</code>	<code>asap2userlib.tlc</code>

File Section	Contents of asap2main.tlc	TLC File Containing Function Definition
- COMPU_METHODS	...WriteCompuMethod()	asap2userlib.tlc
/end MODULE	/end MODULE	
File footer/tail	%<ASAP2UserFcnWriteFileTail(>	asap2userlib.tlc

The Real-Time Workshop Development Process

Introduction (p. C-2)

A Next-Generation Development Tool
(p. C-3)

How MathWorks Tools Streamline
Development (p. C-12)

Code Formats (p. C-18)

An Open and Extensible Environment
(p. C-31)

What Real-Time Workshop provides

How Real-Time Workshop streamlines software design
and development

Maximizing the efficiency of software design,
development and deployment

How Real-Time Workshop saves time and effort by
packaging code in target-specific ways

Real-Time Workshop features you can customize and
elaborate

Introduction

The primary features of Real-Time Workshop are

- **Simulink Code Generator:** Automatically generates C code from your Simulink model.
- **Make Process:** Real-Time Workshop's user-extensible make process lets you create your own production or rapid prototyping target.
- **Simulink External Mode:** External mode enables communication between Simulink and a model executing on a real-time test environment, or in another process on the same machine. External mode lets you perform real-time parameter tuning and data viewing using Simulink as a front end.
- **Targeting Support:** Using the Real-Time Workshop bundled targets, you can build systems for several environments. The generic real-time and embedded real-time targets provide a framework for developing customized rapid prototyping or production target environments. In addition to the bundled targets, the Real-Time Windows Target and/or the xPC Target let you turn a PC of any form factor into a rapid prototyping target, or a small to medium volume production target. The Real-Time Workshop Embedded Coder provides many more targeting, and can be supplemented by turnkey embedded target solutions such as The Embedded Target for Motorola® MPC555 and the Embedded Target for TI C6000 DSP Platform.
- **Rapid Simulations:** Using Simulink Accelerator (part of the Simulink Performance Tools product), S-Function Target, or Rapid Simulation Target, you can accelerate your simulations by 5 to 20 times on average. Executables built with these targets bypass Simulink normal interpretive simulation modes, which must handle all configurations of each basic modeling primitive. The code generated by Simulink Accelerator, S-Function Target, or Rapid Simulation Target is optimized to execute only the algorithms used in your specific model. In addition, these targets apply many optimizations, such as eliminating ones and zeros in computations for filter blocks.

A Next-Generation Development Tool

The MathWorks tools, including Simulink and Real-Time Workshop, are revolutionizing the way embedded systems are designed. Simulink is a very high-level language (VHLL) — a next-generation programming language. A brief look at the history of dynamic and embedded system design methodologies reveals a steady progression toward higher-level design tools and processes:

- **Design -> analog components:** Before the introduction of microcontrollers, design was done on paper and realized using analog components.
- **Design -> hand written assembly -> early microcontrollers:** In the early microprocessor era, design was done on paper and realized by writing assembly code and placing it on microcontrollers. Today, very low-end applications still use assembly language, but advancements in Real-Time Workshop and C compiler technology are obsolescing such techniques.
- **Design -> high-level language (HLL) -> object code -> microcontroller:** The advent of efficient HLL compilers led to the realization of paper designs in languages such as C. HLL code, transformed to assembly language by a compiler, was then placed on a microcontroller. In the early days of high-level languages, programmers often inspected the machine-generated assembly code produced by compilers for correctness. Today, it is taken for granted that the assembly code is correct.
- **Design -> modeling tool -> manual HLL coding -> object code -> microcontroller:** When design tools such as Simulink appeared, designers were able to express system designs graphically and simulate them for correctness. While this process saved considerable time and improved performance, designs were still translated to C code manually before being placed on a microcontroller. This translation process was both time consuming and error prone.
- **Design -> Simulink -> Real-Time Workshop (automatic code generation) -> object code -> microcontroller:** With the addition of Real-Time Workshop, Simulink itself becomes a VHLL. Modeling constructs in Simulink are the basic elements of the language. Real-Time Workshop then compiles models to produce C code. This machine-generated code is produced quickly and correctly. The manual process of transforming designs to code

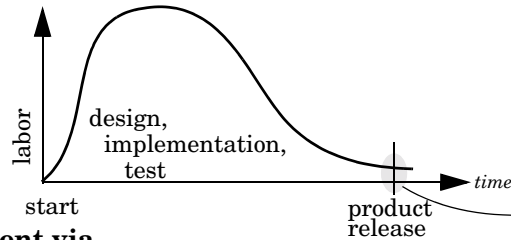
has now been eliminated, yielding significant improvements in system design.

The Simulink code generator included within Real-Time Workshop is a next-generation graphical block diagram compiler. Real-Time Workshop has capabilities beyond those of a typical HLL compiler. Generated code is highly readable and customizable. It is normally unnecessary to read the object code produced by the HLL compiler. You can use Real-Time Workshop in a wide variety of applications, improving your design process.

Key Features

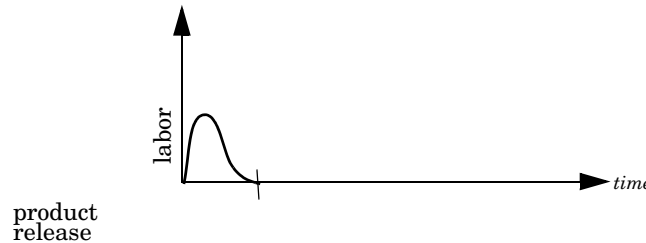
The general goal of the MathWorks tools, including Real-Time Workshop, is to enable you to *accelerate your design process while reducing cost, decreasing time to market, and improving quality.*

Traditional development:



Area under curve indicates the development cost.

Development via the MathWorks tools:



In traditional development practices products often ship before they are completely tested, resulting in a product with defects.

Traditional development practices tend to be very labor intensive. Poor tools often lead to a proliferation of *ad hoc* software projects that fail to deliver reusable code. With the MathWorks tools, you can focus energy on design and achieve better results in less time with fewer people.

Real-Time Workshop, along with other components of the MathWorks tools, provides

- A rapid and direct path from system design to implementation
- Seamless integration with MATLAB and Simulink
- A simple graphical user interface
- An open and extensible architecture

The following features of Real-Time Workshop enable you to reach the above goal:

- **Code generator for Simulink models**
 - Generates optimized, customizable code. There are several styles of generated code, which can be classified as either embedded (production phase) or rapid prototyping.
 - Supports all Simulink features, including 8, 16, and 32 bit integers and floating-point data types.
 - Fixed-point capabilities built into Simulink blocks and Real-Time Workshop allow for scaling of integer words ranging from 2 to 128 bits. Code generation is limited by the implementation of char, short, int, and long in embedded C compiler environments (usually 8, 16, and 32 bits).
 - Generated code is processor independent. The generated code represents your model exactly. A separate run-time interface is used to execute this code. We provide several example run-time interfaces as well as production run-time interfaces.
 - Supports any single or multitasking operating system. Also supports “bare-board” (no operating system) environments.
 - The Target Language Compiler allows extensive customization of the generated code via TLC scripting.
 - Enables custom code generation for S-functions (user-created blocks) using TLC files, enabling you to embed very efficient custom code into the model’s generated code.
- **Extensive model-based debugging support**
 - External mode enables you to examine what the generated code is doing by uploading data from your target to the graphical display elements in your model. There is no need to use a conventional C debugger to look at your generated code.

- External mode also enables you to tune the generated code via your Simulink model. When you change a parametric value of a block in your model, the new value is passed down to the generated code, running on your target, and the corresponding target memory location is updated. Again, there is no need to use an embedded compiler debugger to perform this type of operation. Your model is your debugger user interface.
- **Integration with Simulink**
 - Code verification. You can generate code for your model and create a standalone executable that exercises the generated code and produces a MAT-file containing the execution results.
 - Generated code contains system/block identification tags to help you identify the block, in your source model, that generated a given line of code. The MATLAB command `hilite_system` recognizes these tags and highlights the corresponding blocks in your model.
 - Support for Simulink data objects lets you define how your signals and block parameters interface to the external world.
- **Rapid simulations**
 - Real-Time Workshop supports several ways to speed up your simulations by creating optimized, model-specific executables.
- **Target support**
 - Turnkey solutions for rapid prototyping substantially reduce design cycles, allowing for fast turnaround of design iterations.
 - Bundled rapid prototyping example targets are provided.
 - Add-on targets (Real-Time Windows Target and xPC Target) for PC-based hardware are available from The MathWorks. These targets enable you to turn a PC with fast, high-quality, low cost hardware into a rapid prototyping system.
 - Supports a variety of third-party hardware and tools, with extensible device driver support.
- **Extensible make process**
 - Allows for easy integration with any embedded compiler and linker.
 - Provides for easy linkage with your hand-written supervisory or supporting code.

- **Real-Time Workshop Embedded Coder**

- Customizable, portable, and readable C code that is designed to be placed in a production embedded environment.
- Software-in-the-loop. With Real-Time Workshop Embedded Coder, you can generate code for your embedded application and bring it back into Simulink for verification via simulation.
- Web-viewable code generation report describes code modules, analyzes the generated code, and helps to identify code generation optimizations relevant to your program.
- Annotation of the generated code using the Description block property.
- Hooks for external parameter tuning and signal monitoring are provided enabling easy interfacing of the generated code in your real-time system.

Benefits

You can benefit by using Real-Time Workshop in the following applications. This is not an exhaustive list, but a general survey:

- **Production Embedded Real-Time Applications**

Real-Time Workshop lets you generate, cross-compile, link, and download production C code for real-time systems, such as controllers or digital signal processing (DSP) applications, onto your target processor directly from Simulink. You can customize the generated code by inserting S-functions into your model and specifying, via the Target Language Compiler, what the generated code should look like. Using the optimized, automatically generated code, you can focus your coding efforts on specific features of your product, such as device drivers and general device interfacing.

- **Rapid Prototyping**

As a rapid prototyping tool, Real-Time Workshop enables you to implement your embedded systems designs quickly, without lengthy hand-coding and debugging. Rapid prototyping is typically used in the software/hardware integration and testing phases of the design cycle enabling you to

- Conceptualize solutions graphically in a block diagram modeling environment.
- Evaluate system performance early on - prior to laying out hardware, coding production software, or committing to a fixed design.

- Refine your design by rapid iteration between algorithm design and prototyping.
- Tune parameters while your real-time model runs, using Simulink operating in external mode as a graphical front end.

You can use Real-Time Workshop to generate downloadable, targeted C code that runs on top of a real-time operating system (RTOS). Alternatively, you can generate code to run on the bare hardware at interrupt level, using a simple rate monotonic scheduling executive that you create from examples provided with Real-Time Workshop. There are many rapid prototyping targets provided; or you can create your own.

During rapid prototyping, the generated code is fully instrumented enabling direct access via Simulink external mode for easy monitoring and debugging. The generated code contains a data structure that encapsulates the details of your model. This data structure is used in the bidirectional connection to Simulink running in external mode. Using Simulink external mode, you can monitor signal and tune parameters to further refine your model in rapid iterations enabling you to achieve desired results quickly.

- Real-Time Simulation

You can create and execute code for an entire system or specified subsystems for hardware-in-the-loop simulations. Typical applications include training simulators, real-time model verification, and prototype testing.

- Turnkey Solutions

Bundled Real-Time Workshop targets and third-party turnkey solutions support a variety of control and DSP applications. The target environments include embedded PC, PCI, ISA, VME, and custom hardware, running off-the-shelf real-time operating systems, or Microsoft Windows. Target system processor architectures include Motorola MC680x0 and PowerPC processors, Intel-80x86 and compatibles, Alpha, and Texas Instruments DSPs. Third-party vendors are regularly adding other architectures. For a current list of third-party turnkey solutions, see the MathWorks Connections Program Web page:

<http://www.mathworks.com/products/connections>.

The open environment of Real-Time Workshop also lets you create your own turnkey solution.

- Intellectual Property Protection

The S-Function Target, in addition to speeding up your simulation, allows you to protect your intellectual property: the designs and algorithms embodied in your models. Using the S-Function Target, you can generate and distribute binaries from your models or subsystems. End users have access to the interface, but not to the body, of your algorithms.

- Rapid Simulations

The MathWorks tools can be used in the design of most dynamic systems. Generally Simulink is either used to model a high-fidelity dynamic system (e.g., an engine) or a real-time system (such as an engine controller or a signal processing system).

When modeling high-fidelity systems, you can use Real-Time Workshop to accelerate the design process by speeding up your simulations. This is achieved by using one of the following Real-Time Workshop components:

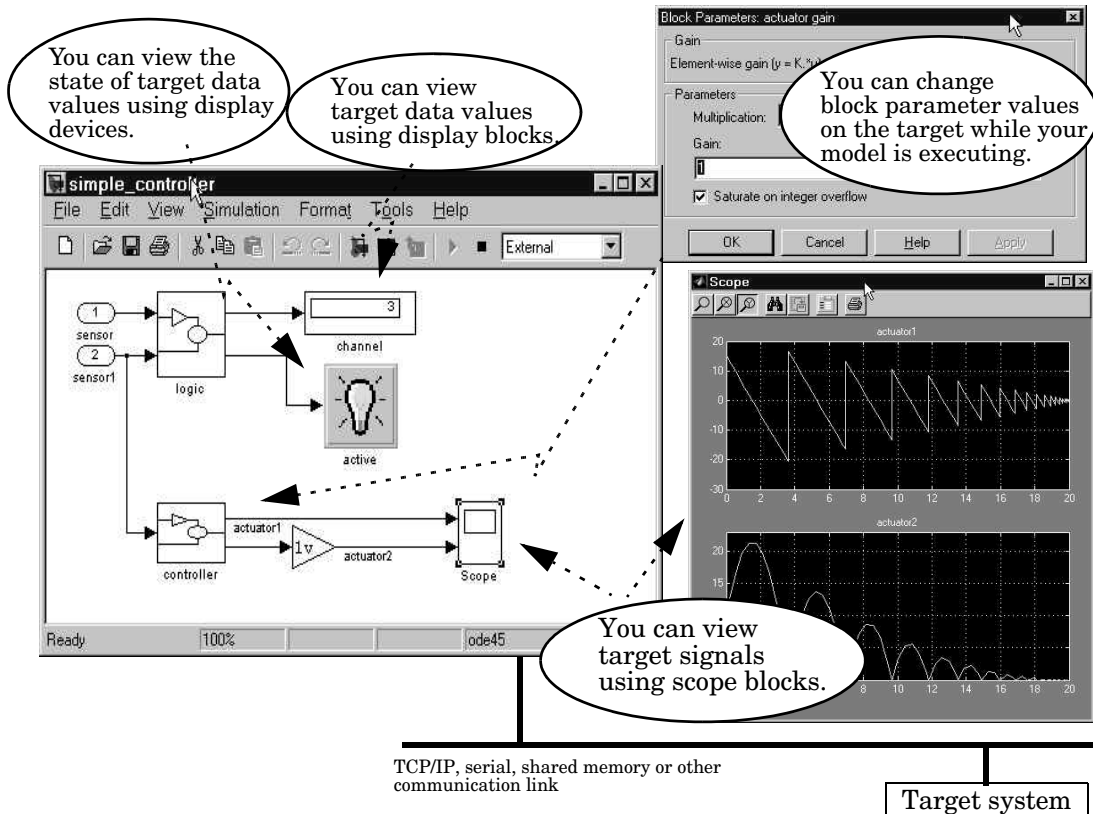
- Simulink Accelerator: Creates a dynamically linked library (MEX-file) from code optimized and generated for your specific model configuration. This executable is used in place of the normal interpretive mode of simulation. Typical speed improvements range from 2 to 8 times faster than normal simulation time. Simulink Accelerator supports both fixed and variable step solvers. Simulink Accelerator is part of the Simulink Performance Tools product.
- Rapid Simulation Target: Creates a stand-alone executable from code optimized and generated for your specific model configuration. This stand-alone executable does not need to interact with a graphics subsystem. Typical speed improvements range from 5 to 20 times faster than normal simulation times. The Rapid Simulation Target is ideal for repetitive (batch) simulations where you are adjusting model parameters or coefficients. The Rapid Simulation Target supports any Simulink solver.
- S-Function Target: This target, like Simulink Accelerator, creates a dynamically linked library (MEX-file) from a model. You can incorporate this component into another model using the Simulink S-function block.

Integration with Simulink

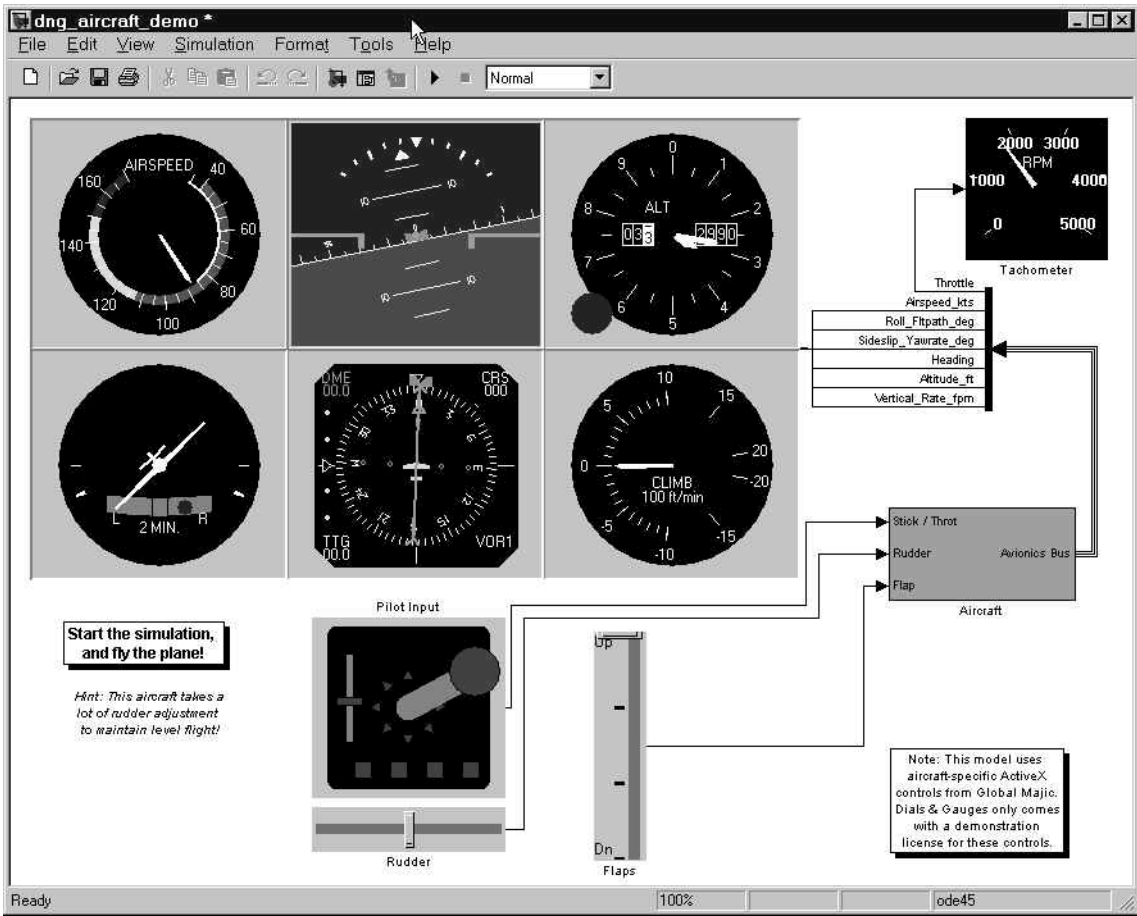
If the Real-Time Workshop target you are using supports Simulink external mode, you can use Simulink as the monitoring/debugging interface for the generated code. With external mode, you can

- Change parameters via the block dialogs, dials, gauges, and the `set_param` MATLAB command. The `set_param` command lets you interact programmatically with your target.
- View target signals in Scope blocks, Display blocks, general S-Function blocks, and via gauges.

These concepts are illustrated by Figure and Figure .



Signal Viewing and Parameter Tuning in External Mode



Simulink model

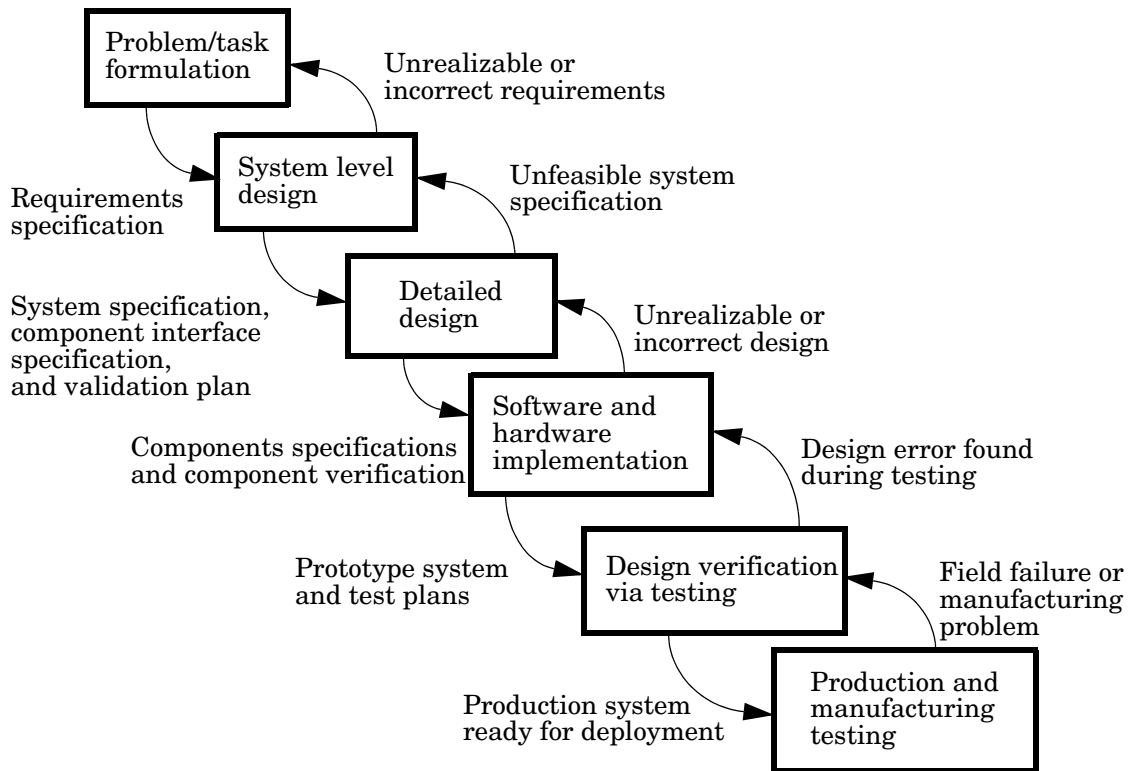
TCP/IP, serial, shared memory or other communication link

Target system

Dials & Gauges Blocks Provide Front End to Target System

How MathWorks Tools Streamline Development

Figure is a high-level view of a traditional development process *without* the MathWorks tools.



Traditional Development Process Without MathWorks Tools

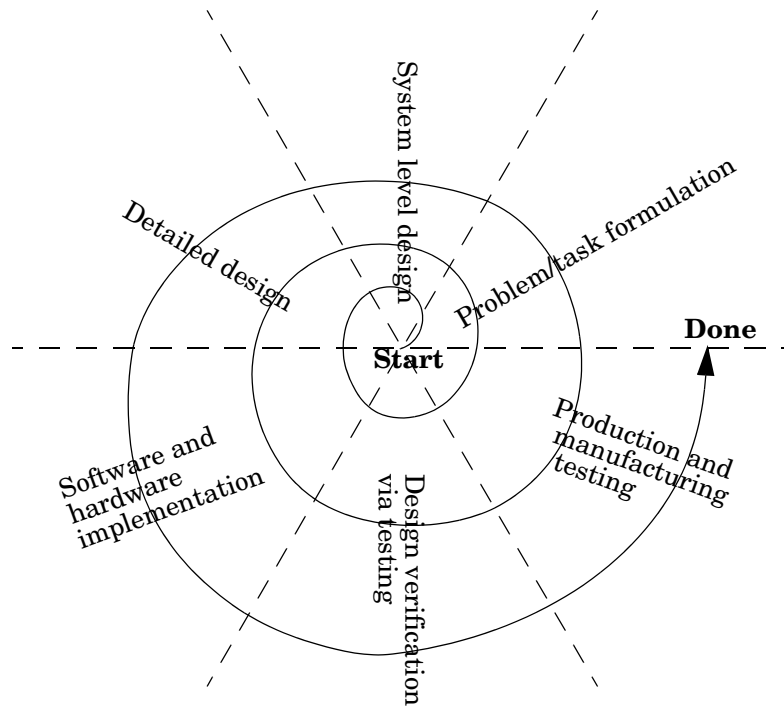
In Figure , each block represents a work phase. Documents are used to coordinate the different work phases. In this environment, it is easy to go back one work phase, but hard to go back multiple work phases. In this environment, design engineers (such as control system engineers or signal processing engineers) are not usually involved in the prototyping phase until

many months after they have specified the design. This can result in poor time to market and inferior quality.

In this environment, different tools are used in each phase. Designs are communicated via paper. This enforces a serial, rather than an iterative, development process. Developers must reenter the result of the previous phase before they can begin work on a new phase. This leads to miscommunication and errors, resulting in lost work hours. Errors found in later phases are very expensive and time consuming to correct. Correction often involves going back several phases. This is difficult because of the poor communication between the phases.

The MathWorks does not suggest or impose a development process. The MathWorks tools can be used to complement any development process. In the above process, use of our tools in each phase can help eliminate paper work.

Our tools also lends itself well to the spiral design process shown in Figure .



Spiral Design Process

Using the MathWorks tools, *your model represents your understanding of your system*. This understanding is passed from phase to phase in the model, reducing the need to go back to a previous phase. In the event that rework is necessary in a previous phase, it is easier to transition back one or more phases, because the same model and tools are used in all phases.

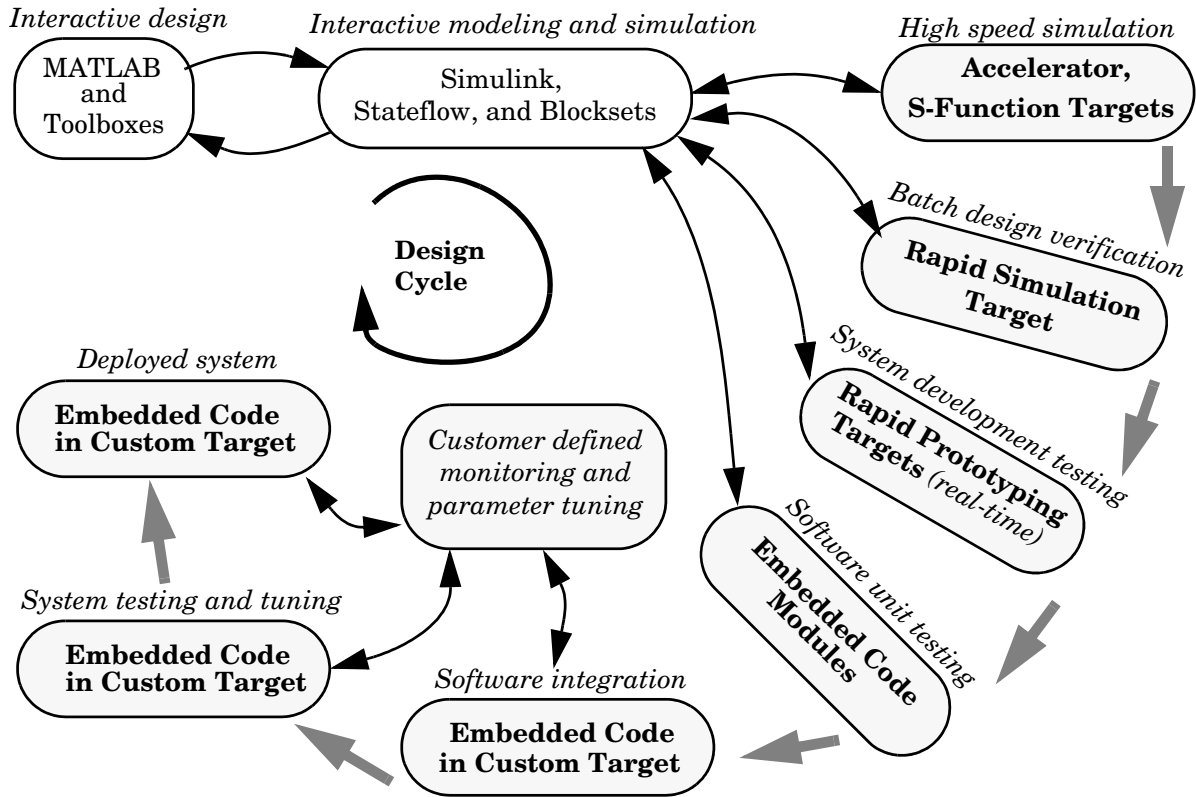
A spiral design process iterates quickly between phases, enabling engineers to work on innovative features. The only way to do this cost effectively is to use tools that make it easy to move from one phase to another. For example, in a matter of minutes a control system engineer or a signal processing engineer can validate an algorithm on a real-world rapid prototyping system. The spiral process lends itself naturally to parallelism in the overall development process. You can provide early working models to validation and production groups,

involving them in your system development process from the start. This helps compress the overall development cycle while increasing quality.

Another advantage of the MathWorks tools is that it enables people to work on tasks that they are good at and enjoy doing. For example, control system engineers specialize in design control rules, while embedded system engineers enjoy assembling systems consisting of hardware and low-level software. It is possible to have very talented people perform different roles, but it is not efficient. Embedded system engineers, for example, are rewarded by specifying and building the hardware and creating low-level software such as device drivers, or real-time operating systems. They do not find data entry operations, such as the manual conversion of a set of equations to efficient code, to be rewarding. This is where the MathWorks tools shines. The equations are represented as models and Real-Time Workshop converts them to highly efficient code ready for deployment.

Role of the MathWorks Tools in Your Development Process

The following figure outlines where the MathWorks tools, including Real-Time Workshop, helps you in your development process.



Roles of MathWorks Tools in Software Design

Early in the design phase, you will start with MATLAB and Simulink to help you formulate your problems and create your initial design. Real-Time Workshop helps with this process by enabling high-speed simulations via Simulink Accelerator (also part of Simulink Performance Tools), and the S-function Target for componentization and model speed-up.

After you have a functional model, you may need to tune your model's coefficients. This can be done quickly using Real-Time Workshop Rapid Simulation Target for Monte-Carlo type simulations (varying coefficients over many simulations).

After you've tuned your model, you can move into system development testing by exercising your model on a rapid prototyping system such as the Real-Time Windows Target or the xPC Target. With a rapid prototyping target, you connect your model to your physical system. This lets you locate design flaws or modeling errors quickly.

After your prototype system is created, you can use the Real-Time Workshop Embedded Coder to create embeddable code for deployment on your custom target. The signal monitoring and parameter tuning capabilities enable you to easily integrate the embedded code into a production environment equipped with debugging and upgrade capabilities.

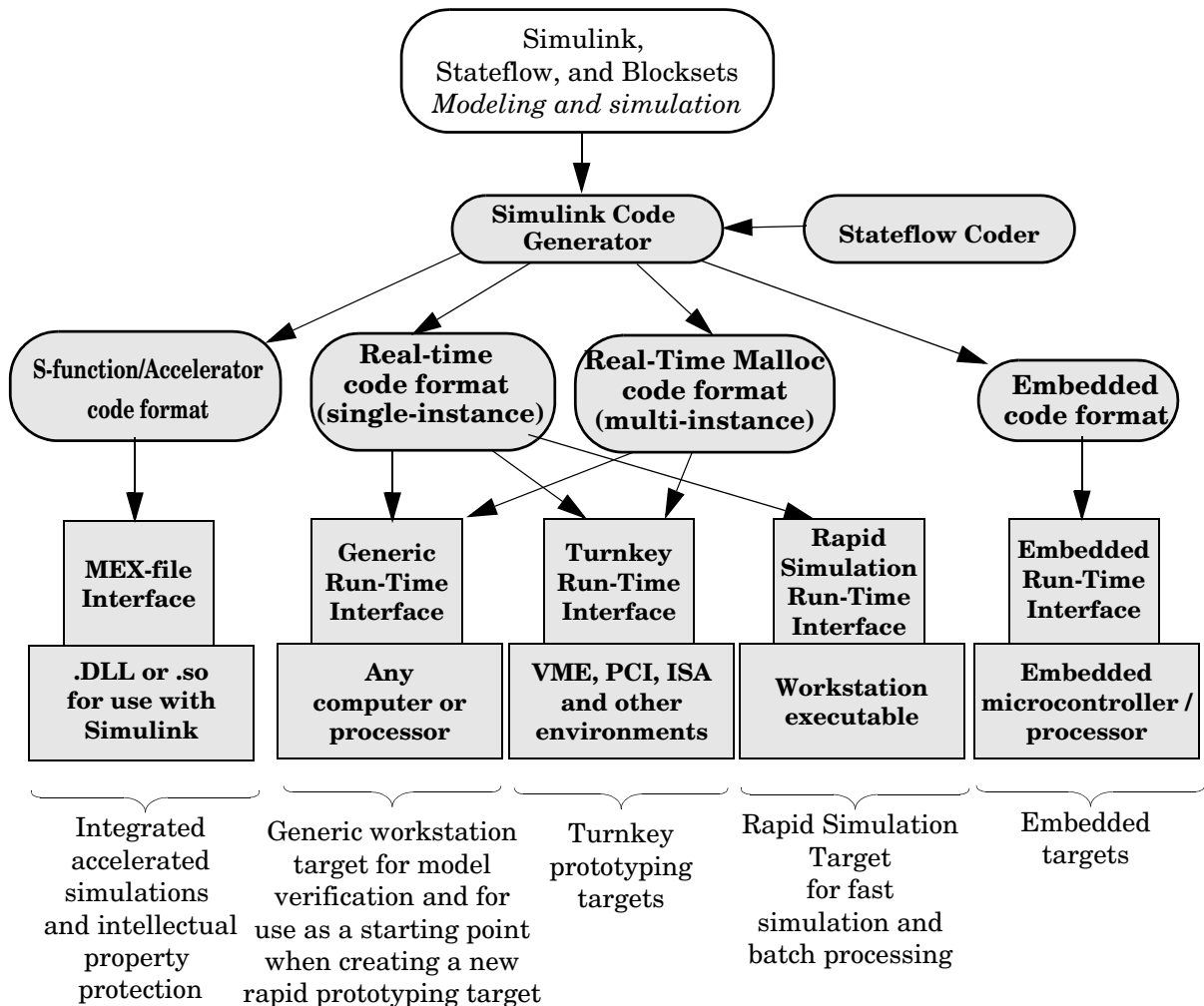
Code Formats

The Real-Time Workshop code generator transforms your model to HLL code. Real-Time Workshop supports a variety of code formats designed for different execution environments, or targets.

In the traditional embedded system development process, an engineer develops an algorithm (or equations) to be implemented in an embedded system. These algorithms are manually converted to a computer language such as C. This translation process, usually done by an embedded system engineer, is much like data entry.

Using Simulink to specify the algorithm (or equations), and Real-Time Workshop to generate corresponding code, engineers can bypass this redundant translation step. This enables embedded system engineers to focus on the key issues involved in creating an embedded system: the hardware configuration, device drivers, supervisory logic, and supporting logic for the model equations. Simulink itself is the programming language that expresses the algorithmic portion of the system.

The Simulink code generator provided with Real-Time Workshop is an open “graphical compiler” supporting a variety of code formats. The relationship between code formats and targets is shown below.



Relationship Between Code Formats and Targets

S-Function/Accelerator Code Format

This code format, used by the S-function Target and Simulink Accelerator, generates code that conforms to Simulink C MEX S-function API.

Real-Time Code Format

The real-time code format is ideally suited for rapid prototyping. This code format (C only) supports increased monitoring and tuning capabilities, enabling easy connection with external mode. Real-time code format supports continuous-time models, discrete-time singlerate or multirate models, and hybrid continuous-time and discrete-time models. Real-time code format supports both inlined and noninlined S-functions. Memory allocation is declared statically at compile time.

Real-Time Malloc Code Format

The real-time malloc code format is similar to the real-time code format. The primary difference is that the real-time malloc code format declares memory dynamically. This supports multiple instances of the same model, with each instance including a unique data set. Multiple models can be combined into one executable without name clashing. Multiple instances of a given model can also be created in one executable.

Embedded Code Format

The embedded code format is designed for embedded targets. The generated code is optimized for speed, memory usage, and simplicity. Generally, this format is used in deeply embedded or deployed applications. There are no dynamic memory allocation calls; all persistent memory is statically allocated.

Real-Time Workshop can generate either C code in the embedded code format. Generating embedded code format requires the Real-Time Workshop Embedded Coder, a separate add-on product for use with Real-Time Workshop.

The embedded code format provides a simplified calling interface and reduced memory usage. This format manages model and timing data in a compact real-time model data structure. This contrasts with the other code formats, which use a significantly larger data structure to manage the generated code.

The embedded code format improves readability of the generated code, reduces code size, and speeds up execution. The embedded code format supports all discrete-time singlerate or multirate models.

The embedded code format supports both inlined and noninlined S-functions.

Target Environments

Real-Time Workshop supports many target environments. These include ready-to-run configurations and third-party targets. You can also develop your own custom target.

This section begins with a list of available target configurations. Following the list, we summarize the characteristics of each target.

Available Target Configurations

Target Configurations Bundled with Real-Time Workshop. The MathWorks supplies the following target configurations with Real-Time Workshop:

- Generic Real-Time (GRT) Target
- Rapid Simulation Target
- S-function Target
- Tornado Real-Time Target

Target Configurations Bundled with Real-Time Workshop Embedded Coder. The MathWorks supplies the following target configuration with Real-Time Workshop Embedded Coder (a separate product from Real-Time Workshop):

- Real-Time Workshop Embedded Coder ERT Target

Turnkey Rapid Prototyping Target Products. These self-contained solutions (separate products from Real-Time Workshop) include:

- Real-Time Windows Target
- Embedded Target for Motorola MPC555 (requires Real-Time Workshop Embedded Coder)
- Embedded Target for Infineon C166® Microcontrollers (requires Real-Time Workshop Embedded Coder)
- xPC Target

DSP Target Products. See *Developer's Kit for Texas Instruments DSP User's Guide* for information on this target:

- Texas Instruments TMS320C6701 Evaluation Module Target
- Embedded Target for TI C6000 DSP Platform

Third-Party Targets. Numerous software vendors have developed customized targets for Real-Time Workshop. For an up-to-date listing of third-party targets, visit the MathWorks Connections Program Web page at <http://www.mathworks.com/products/connections>

View **Third-Party Solutions by Product Type**, and then select Real-Time Workshop Target from the drop-down list.

Custom Targets. Typically, to target custom hardware, you must write a harness (main) program for your target system to execute the generated code, and I/O device drivers to communicate with your hardware. You must also create a system target file and a template makefile.

Real-Time Workshop supplies generic harness programs as starting points for custom targeting. See “*Developing Embedded Targets with Real-Time Workshop Embedded Coder*” in the Real-Time Workshop Embedded Coder documentation for the information you will need to develop a custom target.

Rapid Simulation Target

Rapid Simulation Target (RSIM) consists of a set of target files for non-real-time execution on your host computer. RSIM enables you to use Real-Time Workshop to generate fast, stand-alone simulations. RSIM allows batch parameter tuning and downloading of new simulation data (signals) from a standard MATLAB MAT-file without the need to recompile the model.

The speed of the generated code also makes RSIM ideal for Monte Carlo simulations. The RSIM target enables the generated code to read and write data from or to standard MATLAB MAT-files. RSIM reads new signals and parameters from MAT-files at the start of simulation.

RSIM enables you to run stand-alone, fixed-step simulations on your host computer or on additional computers. If you need to run 100 large simulations, you can generate the RSIM model code, compile it, and run the executables on 10 identical computers. The RSIM target allows you to change the model parameters and the signal data, achieving significant speed improvements by using a compiled simulation.

S-Function and Accelerator Targets

S-Function Target provides the ability to transform a model into a Simulink S-function component. Such a component can then be used in a larger model. This allows you to speed up simulations and/or reuse code. You can include

multiple instances of the same S-function in the same model, with each instance maintaining independent data structures. You can also share S-function components without exposing the details of the a proprietary source model.

The Accelerator Target is similar to the S-Function Target in that an S-function is created for a model. The Accelerator Target differs from the S-Function Target in that the generated S-function operates in the background. It provides for faster simulations while preserving all existing simulation capabilities (parameter change, signal visualization, full S-function support, etc.).

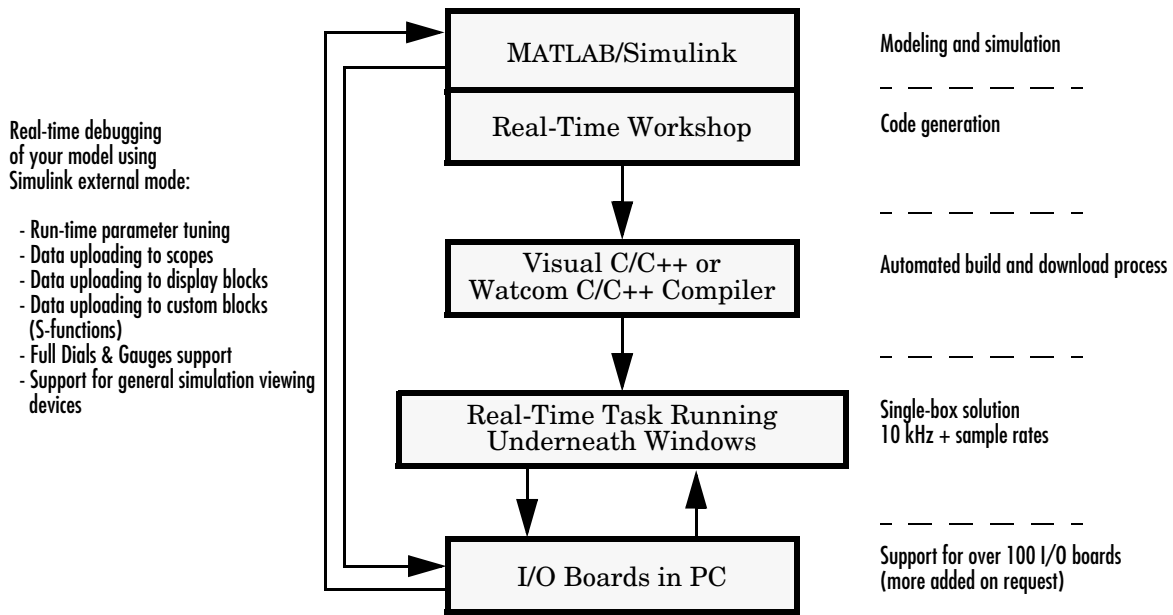
Turnkey Rapid Prototyping Targets

The Real-Time Windows Target and the xPC Target are add-on products to Real-Time Workshop. Both of these targets turn an Intel 80x86/Pentium or compatible PC into a real-time system. Both support a large selection of off-the-shelf I/O cards (both ISA and PCI).

With turnkey target systems, all you need to do is install the MathWorks software and a compiler, and insert the I/O cards. You can then use a PC as a real-time system connected to external devices via the I/O cards.

Real-Time Windows Target. The Real-Time Windows Target brings rapid prototyping and hardware-in-the-loop simulation to your desktop. It is the most portable solution available today for rapid prototyping and hardware-in-the-loop simulation when used on a laptop outfitted with a PCMCIA I/O card. The Real-Time Windows Target is ideal since a second PC or other real-time hardware is often unnecessary, impractical or cumbersome.

This picture shows the basic components of the Real-Time Windows Target.



Real-Time Windows (rtwin) Target

As a prototyping environment, the Real-Time Windows Target is exceptionally easy to use, due to tight integration with Simulink and external mode. It is much like using Simulink itself, with the added benefit of gaining real-time performance and connectivity to the real world through a wide selection of supported I/O boards. You can control your real-time execution with buttons located on the Simulink toolbar. Parameter tuning is done interactively, by simply editing Simulink blocks and changing parameter values. For viewing signals, the Real-Time Windows Target uses standard Simulink Scope blocks, without any need to alter your Simulink block diagram. Signal data can also be logged to a file or set of files for later analysis in MATLAB.

The Real-Time Windows Target is often called the “one-box rapid prototyping system,” since both Simulink and the generated code run on the same PC. A

run-time interface enables you to run generated code on the same processor that runs Windows. The generated code executes in real time, allowing Windows to execute when there are free CPU cycles. The Real-Time Windows Target supports over 100 I/O boards, including ISA, PCI, CompactPCI, and PCMCIA. Sample rates in excess of 10 to 20 kHz can be achieved on Pentium PCs.

In universities, the Real-Time Windows Target provides a cost effective solution since only a single computer is required. In commercial applications, the Real-Time Windows Target is often used at an engineer's desk prior to taking a project to an expensive dedicated real-time testing environment. Its portability is unrivaled, allowing you to use your laptop as a real-time test bed for applications in the field.

Rapid Prototyping Targets

There are two classes of rapid prototyping targets: those using the real-time code format and those using the real-time malloc code format. These differ in the way they allocate memory (statically versus dynamically). Most rapid prototyping targets use the real-time code format.

We define two forms of rapid prototyping environments:

- **Heterogeneous rapid prototyping** environments use rapid prototyping hardware (such as an Intel-80x86/Pentium or similar processor) that differs from the final production hardware. For example, an Intel-80x86/Pentium or similar processor might be used during rapid prototyping of a system that is eventually deployed onto a fixed-point Motorola microcontroller.
- **Homogeneous rapid prototyping** environments are characterized by the use of similar hardware for the rapid prototyping system and the final production system. The main difference is that the rapid prototyping system has extra memory and/or interfacing hardware to support increased debugging capabilities, such as communication with external mode.

Homogeneous rapid prototyping environments eliminate uncertainty because the rapid prototyping environment is closer to the final production system. However, a turnkey system for your specific hardware may not exist. In this case, you must weigh the advantages and disadvantages of using one of the existing turnkey systems for heterogeneous rapid prototyping, versus creating a homogeneous rapid prototyping environment.

Several rapid prototyping targets are bundled with Real-Time Workshop.

Generic Real-Time (GRT) Target. This target uses the real-time code format and supports external mode communication. It is designed to be used as a starting point when creating a custom rapid prototyping target, or for verifying the generated code on your workstation.

Generic Real-Time Malloc (GRTM) Target. This target is similar to the GRT target but it uses the real-time malloc code format. This format uses the C malloc and free routines to manage all data. With this code format, you can have multiple instances of your model and/or multiple models in one executable.

Tornado Target. The Tornado target uses the real-time or real-time malloc code format. A set of run-time interface files are provided to execute your models on the Wind River System's real-time operating system, VxWorks. The Tornado target supports singletasking, multitasking, and hybrid continuous and discrete-time models.

The Tornado run-time interface and device driver files can also be used as a starting point when targeting other real-time operating system environments. The run-time interface provides full support for external mode, enabling you to take full advantage of the debugging capabilities for parameter tuning and data monitoring via graphical devices.

Embedded Targets

The embedded real-time target is the main component of the Real-Time Workshop Embedded Coder. It consists of a set of run-time interface files that drive code, generated in the embedded code format, on your workstation. This target is ideal for memory-constrained embedded applications. Real-Time Workshop supports generation of embedded code in C.

In its default configuration, the embedded real-time target is designed for use as a starting point for targeting custom embedded applications, and as a means by which you can verify the generated code. To create a custom embedded target, you start with the embedded real-time target run-time interface files and edit them as needed for your application.

In the terminology of Real-Time Workshop, an embedded target is a deeply embedded system. Note that it is possible to use a rapid prototyping target in an embedded (production) environment. This may make more sense in your application.

Code Generation Optimizations

The Simulink code generator included with Real-Time Workshop is packed with optimizations to help create fast and minimal size code. The optimizations are classified either as cross-block optimizations, or block specific optimizations. Cross-block optimizations apply to groups of blocks or the general structure of a model. Block specific optimizations, of which there are many, are handled locally by the object generating code for a given block. Target Language Compiler technology generates very tight and fast code for each block in your model.

The following sections discuss some of the cross-block optimizations.

Multirate Support

One of the more powerful features of Simulink is its implicit support for multirate systems. The ability to run different parts of a model at different rates guarantees optimal use of the target processor. In addition, Simulink enforces correctness by requiring that you create your model in a manner that guarantees deterministic execution. The Real-Time Workshop Embedded Coder provides enhanced support for multirate and multitasking systems.

Inlining S-Function Blocks for Optimal Code

The ability to add blocks to Simulink via S-functions is enhanced by the Target Language Compiler. You can create blocks that embed the minimal amount of instructions into the generated code. For example, if you create a device driver using an S-function, you can have the generated code produce one line for the device read, as in the following code fragment:

```
model_outputs(void)
{
    .
    .
    model_B.deviceout = READHW(); /* Macro to read hw device using
    .                               assembly code */
    .
}
```

Note that the generic S-function API is suitable for any basic block-type operation.

Loop Rolling Threshold

The code generated for blocks can contain for loops, or the loop iterations can be “flattened out” into inline statements. For example, the general gain block equation is

```
for (i = 0; i < N; i++) {  
    y[i] = k[i] * u[i];  
}
```

If N is less than a specified roll threshold, Real-Time Workshop expands out the for loop, otherwise Real-Time Workshop retains the for loop.

Tightly Coupled Optimal Stateflow Interface

The generated code for models that combine Simulink blocks and Stateflow charts is tightly integrated and very efficient.

Stateflow Optimizations

The Stateflow Coder contains a large number of optimizations that produce highly readable and very efficient generated code.

Inlining of Systems

In Simulink, a system starting at a nonvirtual subsystem boundary (e.g. an enabled, triggered, enabled and triggered, function-call, or atomic subsystem) can be inlined by selecting the **RTW inline subsystem** option from the subsystem block properties dialog. The default action is to inline the subsystem, unless it is a function-call subsystem with multiple callers.

Block I/O Reuse

Consider a model with a D/A converter feeding a gain block (for scaling), then feeding a transfer function block, then feeding a A/D block. If all signals refer to the same memory location, then less memory will be used. This is referred to as block I/O reuse. It is a powerful optimization technique for re-using memory locations. It reduces the number of global variables, speeding up execution and reducing the memory usage of the generated code.

Declaration of Block I/O Variables in Local Scope

If input/output signal variables are not used across functions, then they can be placed in local scope. This optimization technique also speeds up execution and reduces the memory usage of the generated code.

Inlining of Parameters

If you select the **Inline parameters** option, the numeric values of block parameters that represent coefficients are embedded in the generated code. If **Inline parameters** is off, block parameters that represent coefficients can be changed while the model is executing.

Note that it is possible to specify which parameters to tune using the **Model Parameter Configuration** dialog box, and to tune selected parameters used in referenced models by declaring them as Model block parameter arguments.

Inlining of Invariant Signals

An invariant signal is a block output signal that does not change during Simulink simulation. For example, the output of a sum block that is fed by two constants cannot change. When **Inline invariant signals** is selected on the **Real-Time Workshop Configuration Parameters** dialog pane, a single numeric value is placed in the generated code to represent the output value of the sum block. The **Inline invariant signals** option is available when the **Inline parameters** option is on.

Block Reduction Optimizations

Real-Time Workshop can detect block patterns (e.g. an accumulator represented by a constant, sum and a delay block) and reduce these patterns to a single operation, resulting in very efficient generated code.

Creation of Contiguous Signals to Speed Block Computations

Some block algorithms (for example a matrix multiply) can be implemented more efficiently if the signals entering the blocks are contiguous. Noncontiguous signals occur because of the handling of virtual blocks. For example, the output of a Mux block is noncontiguous. When this class of block requires a contiguous signal, Simulink will insert (if needed) a copy block operator to make the signal contiguous. This results in better code efficiency.

Support for Noncontiguous Signals by Blocks

Noncontiguous signals occur because of the block virtualization capabilities of Simulink. For example, the output of a Mux block is generally a noncontiguous signal (i.e., the output signal consists of signals from multiple sources). General blocks in Simulink support this behavior by generating very efficient code to handle each different signal source in a noncontiguous signal.

Data Type Support

Simulink models support a wide range of data types. You can use double precision values to represent real-world values and then when needed use integers or Booleans for discrete valued signals. You can also use fixed-point (integer scaling) capabilities to target models for fixed-point embedded processors. The wide selection of data types in Simulink models enables you to realize your models efficiently.

Frame Support

In signal processing, a frame of data represents time sampled sequences of an input. Many devices have support in hardware for collecting frames of data. With Simulink and the Signal Processing Blockset, you can use frames and perform frame based operations on the data. Frames are a very efficient way of handling high frequency signal processing applications.

Matrix Support

Most blocks in Simulink support the use of matrices. This enables you to create models that represent high levels of abstractions and produce very efficient generated code.

Virtualization of Blocks

Nearly half of the blocks in a typical model are connection type blocks (e.g. Virtual Subsystem, Inport, Outport, Goto, From, Selector, Bus Selector, Mux, Demux, Ground, and Terminator). These blocks are provided to enable you to create complex models with your desired levels of abstraction. Simulink treats these blocks as virtual, meaning that they impose no overhead during simulation or in the generated code.

An Open and Extensible Environment

The Simulink and Real-Time Workshop model-based software development environment is extensible in several ways.

Custom Code Support

S-functions are dynamically linked objects (.dll or .so) that bind with Simulink to extend the modeling environment. By developing S-functions, you can add custom block algorithms to Simulink. Such S-functions provide supporting logic for the model. S-functions are flexible, allowing you to implement complex algorithmic equations or basic low-level device drivers. Real-Time Workshop support for S-functions includes the ability to inline S-function code directly into the generated code. Inlining, supported by the Target Language Compiler, can significantly reduce memory usage and calling overhead.

Support for Supervisory Code

The generated code implements an algorithm that corresponds exactly to the algorithm defined in your model. With the embedded code format, you can call the generated model code as a procedure. This enables you to incorporate the generated code into larger systems that decide when to execute the generated code. Conceptually, you can think of the generated code as set of equations, wrapped in a function called by your supervisory code. This facilitates integration of model code into large existing systems, or into environments that consist of more than signal-flow processing (Simulink) and state machines (Stateflow).

Monitoring and Parameter Tuning APIs

External mode provides a communication channel for interfacing the generated code running on your target with Simulink. External mode lets you use Simulink as a debugging front end for an executing model. Typically, the external mode configuration works in conjunction with either the real-time code format or the real-time malloc code format.

Real-Time Workshop provides other mechanisms for making model signals and block parameters visible to your own monitoring and tuning interfaces. These mechanisms, suitable for use on all code formats, include:

- The **Model Parameter Configuration** dialog box, where you declare how to allocate memory for variables that are used in your model. For example, if a Gain block contains the variable *k*, you can declare *k* as an external variable, a pointer to an external variable, a global variable, or let Real-Time Workshop decide where and how to declare the variable.

The **Model Parameter Configuration** feature enables you to specify block parameters as tunable or global. This gives your supervisory code complete access to any block parameter variables that you may need to alter while your model is executing. You can also use this feature to interface parameters to specific constant read-only memory locations.

- You can mark signals in your model as *test points*. Declaring a test point indicates that you may want to see the signal's value while the model is executing. After marking a signal as a test point, you specify how the memory for the signal is to be allocated. This gives your supervisory code complete read-only access to signals in your model, so that you can monitor the internal workings of your model.
- C and Target Language Compiler APIs provide another form of access to the signals and parameters in your model. The Target Language Compiler API is a means to access the internal signals and parameters during code generation. With this information, you can generate monitoring/tuning code that is optimized specifically for your model or target.

Interrupt Support

Interrupt blocks enable you to create models that handle synchronous and asynchronous events, including interrupt service routines (ISRs), hardware-generated interrupts, and asynchronous read and write operations. The blocks provided work with the Tornado target. You can use these blocks as templates when creating new interrupt blocks for your target environment. Interrupt blocks include

- Asynchronous Interrupt block
- Task Synchronization block
- Asynchronous Buffer block (read)
- Asynchronous Buffer block (write)
- Asynchronous Rate Transition block

application modules	With respect to Real-Time Workshop program architecture, these are collections of programs that implement functions carried out by the system-dependent, system-independent, and application components.
atomic subsystem	Subsystem whose blocks are executed as a unit before moving on. Conditionally executed subsystems are atomic, and atomic subsystems are nonvirtual. Unconditionally executed subsystems are virtual by default, but can be designated as atomic. Real-Time Workshop can generate reusable code only for nonvirtual subsystems.
base sample rate	Fundamental sample time of a model; in practice, limited by the fastest rate at which a processor's timer can generate interrupts. All sample times must be integer multiples of the base rate.
block I/O structure (model_B)	Global data structure for storing 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. By default, Simulink and Real-Time Workshop try to reduce the size of the <i>model_B</i> structure by reusing the entries in the <i>model_B</i> structure and making other entries local variables.
block target file	File that describes how a specific Simulink block is to be transformed to a language such as C, based on the block's description in the Real-Time Workshop file (<i>model.rtw</i>). Typically, there is one block target file for each Simulink block.
code reuse	Optimization whereby code generated for identical nonvirtual subsystems is collapsed into one function that is called for each subsystem instance with appropriate parameters. Code reuse, along with <i>expression folding</i> , can dramatically reduce the amount of generated code.
configuration	Set of attributes for a model which defines parameters governing how a model simulates and generates code. A model can have one or more such configuration sets, and users can switch between them to change code generation targets or to modify the behavior of models in other ways.
configuration component	Named element of a configuration set. Configuration components encapsulate settings associated with the Solver , Data Import/Export , Optimization , Diagnostics , Hardware Implementation , Model Referencing , and Real-Time Workshop panes in the Configuration Parameters dialog. A component may contain sub-components.
embedded real-time (ERT) target	Target configuration that generates model code for execution on an independent embedded real-time system. Requires Real-Time Workshop Embedded Coder.

expression folding

Code optimization technique that minimizes the computation of intermediate results at block outputs and the storage of such results in temporary buffers or variables. It can dramatically improve the efficiency of generated code, achieving results that compare favorably with hand-optimized code.

file extensions

The table below lists the file extensions associated with Simulink, the Target Language Compiler, and Real-Time Workshop.

Extension	Created by	Description
.c	Target Language Compiler	The generated C code
.h	Target Language Compiler	C include header file used by the .c program
.mdl	Simulink	Contains structures associated with Simulink block diagrams
.mk	Real-Time Workshop	Makefile specific to your model that is derived from the template makefile
.rtw	Real-Time Workshop	Intermediate compilation (<i>model.rtw</i>) of a .mdl file used in generating C code
.tlc	The MathWorks and Real-Time Workshop users	Target Language Compiler script files that Real-Time Workshop uses to generate code for targets and blocks
.tmf	Supplied with Real-Time Workshop	Template makefiles
.tmw	Real-Time Workshop	Project marker file inside a build directory that identifies the date and product version of generated code

generic real-time (GRT) target

Target configuration that generates model code for a real-time system, with the resulting code executed on your workstation. (Note that execution is not tied to a real-time clock.) You can use GRT as a starting point for targeting custom hardware.

host system	Computer system on which you create and may compile your real-time application. Also referred to as emulation hardware.
inline	Generally, this means to place something directly in the generated source code. You can inline parameters and S-functions using Real-Time Workshop and the Target Language Compiler.
inlined parameters	(Target Language Compiler Boolean global variable: <code>InLineParameters</code>) The numerical values of the block parameters are hard-coded into the generated code. Advantages include faster execution and less memory use, but you lose the ability to change the block parameter values at run-time.
inlined S-function	An S-function can be inlined into the generated code by implementing it as a <code>.t1c</code> file. The code for this S-function is placed in the generated model code itself. In contrast, noninlined S-functions require a function call to an S-function residing in an external MEX-file.
interrupt service routine (ISR)	Piece of code that your processor executes when an external event, such as a timer, occurs.
loop rolling	(Target Language Compiler global variable: <code>RollThreshold</code>) Depending on the block's operation and the width of the input/output ports, the generated code uses a <code>for</code> statement (rolled code) instead of repeating identical lines of code (flat code) over the signal width.
make	Utility to maintain, update, and regenerate related programs and files. The commands to be executed are placed in a <i>makefile</i> .
makefiles	Files that contain a collection of commands that allow groups of programs, object files, libraries, etc., to interact. Makefiles are executed by your development system's <code>make</code> utility.
model.rtw	Intermediate record file into which Real-Time Workshop compiles the blocks, signals, states, and parameters a model, which the Target Language Compiler reads to generate code to represent the model.
multitasking	Process by which a microprocessor schedules the handling of multiple tasks. In generated code, the number of tasks is equal to the number of sample times in your model. <i>See also</i> pseudomultitasking.
noninlined S-function	In the context of Real-Time Workshop, this is any C-MEX S-function that is not implemented using a customized <code>.t1c</code> file. If you create a C-MEX S-function as

	part of a Simulink model, it is by default noninlined unless you write your own <code>.tlc</code> file that inlines it.
nonreal-time	Simulation environment of a block diagram provided for high-speed simulation of your model. Execution is not tied to a real-time clock.
nonvirtual block	Any block that performs some algorithm, such as a Gain block. Real-Time Workshop generates code for all nonvirtual blocks, either inline or as separate functions and files, as directed by users.
pseudo multitasking	On processors that do not offer <i>multitasking</i> support, you can perform pseudomultitasking by scheduling events on a fixed time-sharing basis.
real-time model data structure	Real-Time Workshop encapsulates information about the root model in the real-time model data structure, often abbreviated as <code>rtM</code> . <code>rtM</code> contains global information related to timing, solvers, and logging, and model data such as inputs, outputs, states, and parameters.
real-time system	Computer that processes real-world events as they happen, under the constraint of a real-time clock, and that can implement algorithms in dedicated hardware. Examples include mobile telephones, test and measurement devices, and avionic and automotive control systems.
Real-Time Workshop target	Set of code files generated by Real-Time Workshop for a standard or custom target, specified by a Real-Time Workshop configuration component. These source files can be built into an executable program that will run independently of Simulink. <i>See also</i> simulation target, configuration.
run-time interface	Wrapper around the generated code that can be built into a stand-alone executable. The run-time interface consists of routines to move the time forward, save logged variables at the appropriate time steps, etc. The run-time interface is responsible for managing the execution of the real-time program created from your Simulink block diagram.
S-function	Customized Simulink block written in C, Fortran, or M-code. Real-Time Workshop can target C-code S-functions as is or users can <i>inline</i> C-code S-functions by preparing TLC scripts for them.
simstruct	Simulink data structure and associated application program interface (API) that enables S-functions to communicate with other entities in models. Simstructs are included in code generated by Real-Time Workshop for noninlined S-functions.

simulation target	Set of code files generated for a model which is referenced by a Model block. Simulation target code is generated into <code>/slprj/sim</code> project directory in the working directory. Also an executable library compiled from these codes that implements a Model block. <i>See also</i> Real-Time Workshop Target.
singletasking	Mode in which a model runs in one task, regardless of the number of sample rates it contains.
stiffness	Property of a problem that forces a numerical method, in one or more intervals of integration, to use a step length that is excessively small in relation to the smoothness of the exact solution in that interval.
system target file	Entry point to the Target Language Compiler program, used to transform the Real-Time Workshop file into target-specific code.
target file	File that is compiled and executed by the Target Language Compiler. The block and system target TLC files used specify how to transform the Real-Time Workshop file (<i>model.rtw</i>) into target-specific code.
Target Language Compiler (TLC)	Program that compiles and executes system and target files by translating a <i>model.rtw</i> file into a target language by means of TLC scripts and template makefiles.
Target Language Compiler program	One or more TLC script files that describe how to convert a <i>model.rtw</i> file into generated code. There is one TLC file for the target, plus one for each built-in block. Users can provide their own TLC files in order to inline S-functions or to wrap existing user code.
target system	Specific or generic computer system on which your real-time application is intended to execute. Also referred to as embedded hardware.
targeting	Process of creating software modules appropriate for execution on your target system.
task identifier (tid)	In generated code, each sample rate in a multirate model is assigned a task identifier (<i>tid</i>). The <i>tid</i> is used by the model output and update routines to control the portion of your model that should execute at a given time step. Single-rate systems ignore the <i>tid</i> . <i>See also</i> base sample rate.
template makefile	Line-for-line makefile used by a make utility. Real-Time Workshop converts the template makefile to a makefile by copying the contents of the template makefile (usually <i>system.tmf</i>) to a makefile (usually <i>system.mk</i>) replacing tokens describing your model's configuration.

virtual block

Connection or graphical block, for example a Mux block, that has no algorithmic functionality. Virtual blocks incur no real-time overhead as no code is generated for them.

work vector

Data structures for saving internal states or similar information, accessible to blocks that may require such work areas. These include state work (`rtDWork`), real work (`rtRWork`), integer work (`rtIWork`), and pointer work (`rtPWork`) structures. For example, the Memory block uses a real work element for each signal.

Symbols

<code>utassert 2-27

A

and Stateflow optimizations 2-26

application modules

 application-specific components 7-35

 definition of 7-23

 system-independent components 7-29

ASAP2 files

 customizing B-12

 data attributes required for B-4

 generating B-6

 structure of B-18

 targets supporting B-3

assertion blocks

 in generated code 2-27

Async interrupt block 16-5

atomic subsystem 4-2

automatic S-function generation 11-12

See also S-function target

B

block reduction optimization 2-18

block states

 Simulink data objects and 5-53

 State Properties dialog and 5-49

 storage and interfacing 5-47

 storage classes for 5-48

 symbolic names for 5-50

blocks

 Async Interrupt 16-5

 depending on absolute time A-1

 Rate Transition 8-12

 scope 2-16

 Task Synchronization 16-15

 to file 2-16

 to workspace 2-16

buffer reuse option 2-23

Build button 2-3

build command 2-3

C

C-API

 files used in 17-5

 generating files 17-3

 introduction 17-2

 mapping to real-time model 17-11

 using for your application 17-12

code format

 choosing 3-9

 embedded C 3-18

 real-time 3-13

 real-time malloc 3-15

 S-function 3-17

code generation 2-1

 and simulation parameters 2-11

 from nonvirtual subsystems 4-2

code generation options

 block reduction 2-18

 Boolean logic signals 2-20

 buffer reuse 2-23

see also signal storage reuse

 Expression folding 9-10

 generate HTML report 2-5

 GRT compatible call interface 3-20

 inline invariant signals 2-23

 inline parameters 2-20

 local block outputs 2-23

see also signal storage reuse

- loop rolling threshold 2-24
- MAT-file variable name modifier 2-44
- retain .rtw file 2-41
- show eliminated statements 2-37
- signal storage reuse 2-22
 - see also* Llocal block outputs
- Solver pane 2-11
- verbose builds 2-41
- Workspace I/O pane 2-13
- code reuse
 - diagnostics for 4-14
 - enabling 4-10
 - presence of blocks preventing 4-11
 - subsystem characteristics preventing 4-11
- code tracing
 - via <CF>hilite_system command 2-45
 - via HTML reports 2-45
- combining models
 - in RTW Embedded Coder target 17-29
 - via grt_malloc target 17-29
- communication
 - external mode 6-2
 - external mode API for 17-18
- Configuration Parameters dialog
 - Real-Time Workshop pane 4-2
- configuration parameters dialog 2-11
 - Data Import/Export pane 2-13
 - Real-Time Workshop pane 2-2
 - Solver options pane 2-11
- context-sensitive help 2-4
- continuous states, integration of 7-28
- Custom Code library
 - overview 14-2

D

- data logging 2-13

- in single- and multitasking models 2-16
- to MAT-files 2-14
- via scope blocks 2-16
- via to file blocks 2-16
- via to workspace blocks 2-16
- Data Store Memory blocks
 - Simulink data objects and 5-58
- data structures in generated code
 - block I/O 7-20
 - block parameters 7-20
 - block states 7-20
 - external inputs 7-20
 - external outputs 7-20
- device driver blocks
 - VxWorks 13-14
- direct index lookup table example 10-23
- direct-index lookup table algorithm 10-22
- directories
 - used in build process 2-68

E

- Embedded MATLAB blocks 2-26
- Euler integration algorithm 7-28
- examples
 - direct index lookup table 10-23
 - multiport S-function 10-18
- Expression folding 9-10
 - configuring options for 9-13
 - in S-Functions 10-46
- External mode
 - baud rates 6-12
- external mode 6-2
 - architecture 6-26
 - blocks compatible with 6-22
 - client-server architecture 17-18
 - command line options for target program 6-33

- communication channel creation 17-18
- communications overview 17-21
- configuration parameter options 6-3
- configuring to use sockets 13-7
- control panel options 6-10
- data archiving options 6-18
- design of 17-18
- download mechanism 6-25
- error conditions 6-35
- ext_comm MEX-file
 - optional arguments to 6-29, 6-31
 - rebuilding 17-27
- host and target systems in 6-2
- limitations of 6-35
- menu and toolbar options 6-5
- parameter downloading options 6-20
- Signal Viewing Subsystems in 6-22
- signals and triggering options 6-13
- target communications options 6-12
- TCP implementation 6-28
- transport layer 17-21
 - using with VxWorks 13-6
- external mode API
 - host source files 17-22
 - implementing transport layer 17-25
 - target source files 17-24
- External Target Interface dialog box
 - MEX-file arguments 6-12

G

- general code appearance options
 - Maximum identifier length 2-38
- generated code
 - operations performed by 7-29
- generated files
 - contents of 2-60

- dependencies among 2-60
- generated s-functions
 - tunable parameters in 11-10

H

- host
 - in external mode 6-2

I

- inlined S-functions 10-18
 - with mdlRTW routine 10-20
- Interrupt handling blocks 16-5
- interrupt service routine
 - under VxWorks 8-3

L

- local block outputs option 2-23

M

- make_rtw 2-9
- MAT-files
 - file naming convention 2-15
 - logging data to 2-14
 - variable names in 2-15
- MATLAB C-5
- matlabroot/simulink/src/sfun_multiport.c
 - example file 10-18
- matlabroot/toolbox/simulink/blocks/tlc_c/
 - sfun_multiport.tlc example file 10-18
- mdlRTW routine
 - writing inlined S-functions 10-20
- MEX S-function wrapper
 - definition 10-9

- model blocks
 - in Real-Time Workshop 4-18
- model code
 - execution of 7-33
- model execution
 - in real time 8-10
 - in Simulink 8-10
 - Simulink vs. real-time 8-9
- Model Parameter Configuration dialog
 - tunable parameters and 5-2
 - using 5-8
- model reference
 - code generation 4-18
 - compatibility of top and referenced models 4-22
 - controlling code generation 4-21
 - inherited sample time and 4-35
 - parameter interfacing 4-25
 - project directory structure and 4-27
 - signal interfacing 4-24
 - subsystem code reuse and 4-40
- model registration function 7-33
- multiple models
 - combining 17-29
- multiport S-function example 10-18
- multitasking
 - model execution 8-5
- multitasking
 - automatic rate transition 8-16
 - building program for 8-8
 - enabling 8-8
 - example model 8-24
 - execution 8-27
 - operation 8-11
 - task identifiers in 8-5
 - task priorities 8-5
 - versus singletasking 8-3
- N**
 - noninlined S-functions 10-7
 - nonvirtual subsystem code generation
 - Auto option 4-3
 - Function option 4-7
 - Inline option 4-5
 - Reusable function option 4-10
 - nonvirtual subsystems
 - atomic 4-2
 - categories of 4-2
 - conditionally executed 4-2
 - modularity of code generated from 4-14
- O**
 - operating system
 - tasking primitives 7-10
 - VxWorks 13-2
 - optimization pane
 - Stateflow and eML options 2-26
- P**
 - parameters
 - interfacing 5-2
 - storage declarations 5-2
 - tunable 5-2
 - tuning 5-2
 - priority
 - of sample rates 8-6
 - of VxWorks tasks 13-11
 - program architecture
 - embedded 7-36
 - initialization functions 7-25
 - main function 7-25
 - model execution 7-27
 - program execution 7-13

- program termination 7-26
- program timing 7-12
- rapid prototyping 7-23
- real-time 7-23
- termination functions 7-33

pseudomultitasking 8-6

R

- rapid prototyping 1-5
 - for control systems 1-9
 - for digital signal processing 1-8
- rapid simulation target
 - batch simulations (Monte Carlo) 12-17
 - command line options 12-8
 - limitations 12-18
 - output filename specification 12-14
 - parameter structure access 12-9
 - signal data file specification 12-11
 - Simulink license checkout 12-3
- Rate Transition block 8-12
- Rate Transition blocks
 - auto-insertion of 8-16
- rate transitions
 - faster to slower 8-17
 - slower to faster 8-20
- real time
 - executing models in 8-10
 - integrating continuous states in 7-28
- real-time malloc target 3-15
 - combining models with 17-29
- Real-time model
 - description 7-30
- Real-Time Workshop
 - open architecture of 1-11
 - user interface 2-2, 4-2, 5-2
- Real-Time Workshop pane

- opening 2-2
- overview 2-2, 4-2
- Target configuration options
 - Browse button 2-4
 - generate code only option 2-10
 - make command field 2-9
 - system target file field 2-5
 - template makefile field 2-9
- target configuration options
 - Build button 2-3
- referenced models
 - code generation incompatibilities 4-22
 - generating code for 4-18
- rsim. *See* rapid simulation target
- rtw_local_blk_outs 5-23
- RTWdata structure
 - inlining an S-function 10-21

S

- sample rate transitions 8-12
 - faster to slower
 - in real-time 8-18
 - in Simulink 8-17
 - slower to faster
 - in real-time 8-20
 - in Simulink 8-20
- sample time
 - overlaps 8-21
- sample time constraints
 - setting for multitasking 8-5
- sample time properties
 - setting for multitasking 8-5
- S-function target 3-17
 - applications of 11-2
 - automatic S-function generation 11-12
 - generating reusable components with 11-4

- intellectual property protection in 11-3
- restrictions 11-16
- tunable parameters in 11-10
- unsupported blocks 11-19
- S-functions
 - API 7-34
 - fully inlined with `mdlRTW` routine 10-20
 - generating automatically
 - inlined 10-18
 - models containing 7-33
 - noninlined 7-33, 10-7
 - that work with Real-Time Workshop 10-2
 - types of 10-3
 - wrapper 10-9
- signal properties 5-31
 - setting via Signal Properties dialog 5-31
- signal storage reuse option 2-22
- Signal Viewing Subsystems 6-22
- `SimStruct` data structure
 - and global registration function 7-33
 - definition of 7-29
- Simulation Parameters dialog
 - Real-Time Workshop pane 5-2
- Simulink C-5
 - interactions with Real-Time Workshop 2-46
 - block execution order 2-49
 - sample time propagation 2-48
 - simulation parameters
 - and code generation 2-11
- Simulink data objects 5-33
 - parameter objects 5-34
 - signal objects 5-38
- Singletasking
 - operation 8-11
- singletasking 8-8
 - building program for 8-8
 - enabling 8-9

- example model 8-24
- execution 8-25
- step size
 - of real-time continuous system 7-28
- StethoScope
 - See* `VxWorks`
- subsystem
 - nonvirtual 4-2
- System target file browser 2-53

T

- target
 - available configurations
 - bundled C-21
 - custom C-22
 - third-party C-22
 - rapid simulation *See* rapid simulation target
 - real-time malloc *See* real-time malloc target
- targets
 - available configurations 2-55
- task identifier (`tid`) 8-5
- Task Synchronization block 16-15
- template makefile
 - compiler-specific 2-71
 - default 2-68
- template makefile options
 - Borland 2-75
 - LCC 2-76
 - UNIX 2-72
 - Visual C/C++ 2-73
 - Watcom 2-74
- tooltips 2-4
- tunable expressions 5-2
 - in masked subsystems 5-14
 - limitations on 5-13

V

VxWorks

- and external mode 13-6
- application overview 13-5
- configuring
 - for external mode (sockets) 13-6
 - makefile template 13-14
- connecting target to Ethernet 13-5
- downloading and running the executable
 - interactively 13-20
- GNU tools for 13-15
- implementation overview 13-12
- program build options 13-15
- program execution 13-21
- program monitoring 13-5
- real-time operating system 13-2
- runtime structure 13-9
- StethoScope code generation option 13-18
- target
 - connecting to 13-20
 - downloading to 13-20
- target CPU 13-5
- tasks created by 13-10
- template makefiles 13-14

W

- wrapper S-functions 10-9